# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*



**1. AGENCY USE ONLY** *(Leave blank)*

**2. REPORT DATE**
[illegible]

**3. REPORT TYPE AND DATES COVERED**
DISSERTATION

**4. TITLE AND SUBTITLE**
AN AUTOMATED FRAMEWORK FOR MANAGING DESIGN COMPEXITY

**5. FUNDING NUMBERS**

**6. AUTHOR(S)**
MAJ JACOBS TIMOTHY M

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
UNIVERSITY OF UTAH

**8. PERFORMING ORGANIZATION REPORT NUMBER**

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
THE DEPARTMENT OF THE AIR FORCE
AFIT/CIA, BLDG 125
2950 P STREET
WPAFB OH 45433

**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
FY99-18

**11. SUPPLEMENTARY NOTES**

**12a. DISTRIBUTION AVAILABILITY STATEMENT**
Unlimited distribution
In Accordance With AFI 35-205/AFIT Sup 1

**12b. DISTRIBUTION CODE**

**13. ABSTRACT** *(Maximum 200 words)*

**14. SUBJECT TERMS**

**15. NUMBER OF PAGES**
140

**16. PRICE CODE**

**17. SECURITY CLASSIFICATION OF REPORT**

**18. SECURITY CLASSIFICATION OF THIS PAGE**

**19. SECURITY CLASSIFICATION OF ABSTRACT**

**20. LIMITATION OF ABSTRACT**

# AN AUTOMATED FRAMEWORK FOR MANAGING DESIGN COMPLEXITY

by

Timothy M. Jacobs

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

The University of Utah

December 1998

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

# SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Timothy M. Jacobs

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair: Elaine Cohen

Samuel Drake

Robert R. Kessler

Richard F. Riesenfeld

Keith A. Shomper

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

# FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of Timothy M. Jacobs in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Elaine Cohen
Chair, Supervisory Committee

Approved for the Major Department

Robert R. Kessler
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

# ABSTRACT


Complexity in modern product design is manifest through the interactions of large numbers of diverse parts and functions, and multiple design disciplines. The intricate web of synergistic relationships necessary to link components together makes it difficult for designers to assimilate or represent such complex designs in their totality.

Since existing CAD software tools provide only limited support for managing complex designs, it is necessary to document and track complexity relationships independent of the actual CAD models. This reduces the level of detail that can be managed, while requiring more work from the design team and increasing the risk of inconsistencies and errors in the design. To better support management of complex designs, this research integrates multiple design representations and the relationships among them into a single organizational framework. Its goal is to provide flexibility for designers to manage and evolve design representations for a variety of design processes and applications.

This research uses design data objects to represent the aggregation hierarchy and relationships between design representations. *Aggregation objects* are introduced to organize the design into a multileveled hierarchy by encapsulating multiple design components or representations into single objects. This organization serves to abstract design information and facilitate understanding. The design effects that result from the synergistic interaction between components are captured in *relationship objects.* Relationship objects eliminate duplicate specification and ensure compatibility between components. Together, aggregation and relationship objects form well-defined boundaries between design entities to facilitate simultaneous design and reuse. Changes to aggregation and relationship objects are captured in *version objects* that record the history of a design as it evolves.

To Mary Beth, Timmy, and Danny

You guys are the greatest!

# CONTENTS

# LIST OF FIGURES

# ACKNOWLEDGEMENTS

My sincerest appreciation and thanks goes to each of the members of my committee for their time and advice. Each of you has helped to shape and improve this research. I especially thank my advisor, Elaine Cohen, for her encouragement and advice, and for helping me finish in the allotted time.

I would like to thank the Air Force, and especially the people at the Air Force Institute of Technology who were critical to my selection, for their wisdom in giving me a chance to earn this degree. This educational program has truly been the opportunity of a life time.

My thanks and encouragement also extend to the students and staff of the *Alpha_1* research group, whose assistance was essential to understanding the intricacies of *Alpha_1* and forming solutions which are useful to the *Alpha_1* design environment. I expect the relationships established here to last for a long time.

I thank my parents, brothers, and sisters for helping me develop the proper attitude and work ethic for completing such a difficult task. Extra special thanks go to my wife, Mary Beth, for her love and support and for taking up the slack whenever things got a little bit hectic. I also thank my sons, Timmy and Danny, for just being so wonderful and providing me the necessary distractions to appreciate the really important things in life.

A lot of other people have also contributed to my success in getting here and completing this degree. I sincerely thank each of you.

This work has been supported in part by the NSF Science and Technology Center for Computer Graphics and Scientific Visualization (ASC-89-20219), and DARPA (F33615-96-C-5621). All opinions, findings, conclusions, or recommendations expressed in this document are those of the author and do not necessarily reflect the views of the sponsoring agencies.

# CHAPTER 1

# INTRODUCTION

## 1.1 Overview

Complexity in modern product design is manifest through large numbers of diverse parts, complicated geometry of individual parts, and multiple functions performed by a single part. As a design evolves, multiple representations of a component are created as additional details are added or different design disciplines are considered. The large number of parts, functionality, and design representations creates a significant management problem. This problem is magnified by the complex relationships between the different components and representations. These relationships, which might include the impact of manufacturing capabilities on geometry or the bearings and forces involved in the rotation of a wheel around an axle, provide additional insight into the overall product that is not available in the individual representations.

Existing CAD systems improve the designer's ability to create large numbers of components and design representations by simplifying the creation and analysis of individual parts and components. These design representations, however, require multiple tools and design formats that must be managed through manual processes or data management tools that work with complete design documents at a high level of granularity. Unfortunately, this large scale granularity does not help with the management of the small granularity changes that typically occur as a design evolves. Instead, designers must manually track and control these changes – a process that consumes valuable design time and increases the potential for introducing errors.

This research introduces an organizational framework for automating the management of complex product designs. The objectives of this framework are to:

- Integrate multiple design components, design functionality, and different design disciplines into a single model.
- Support independent manipulation of design components and relationships at multiple levels of granularity.
- Provide flexibility for designers to manage and evolve design representations for a variety of design processes and applications.

This research realizes its organizational framework through three types of data objects – *aggregations*, *relationships*, and *versions*. Aggregation objects encapsulate multiple design components or representations into a single object that can be independently manipulated by the designer. Aggregations may be nested within other aggregations to organize design information into a hierarchy with multiple levels of detail. Relationship objects specify the interaction between components and views at multiple levels of detail. Relationship objects include synergistic design information, such as additional functionality or fasteners, that results from the interaction of two components. In addition to the aggregation and relationship objects, this framework includes version objects to represent the history of a design as it evolves over time. Associated with the aggregation, relationship, and version objects are automated "software assistants" that facilitate the management and analysis of the design information represented by the framework.

The framework presented in this research has many characteristics that help the user to manage complex designs. By organizing the design into aggregations, the designer works with a limited number of components at a time, making it easier to understand and control the design as it changes over time. By using a single object to capture the synergistic information associated with the interaction between components, compatibility of the interacting parts can be controlled and the information does not have to be duplicated in multiple parts. By isolating components with relationship objects and embedding them into aggregations, designers can control modifications to the components, yet still communicate changes

to the remainder of the model. This isolation also facilitates simultaneous design and reuse.

## 1.2 Background

### 1.2.1 Design Complexity

Complexity in modern design manifests itself in many different ways. Product design models contain a vast quantity of diverse information that is linked together in a variety of configurations. These products are developed over a period of time through an extensive design process. During this process, designers brings together information such as customer needs and scientific and engineering principles. They then form this information into a high-level design model and proceed to evolve this model into a working product design. The designer applies numerous techniques to minimize and manage the complexity of the design process and the product being designed.

Product complexity results from a large number of parts in an assembly, complex geometry or multiple functions within an individual part, and the combination of many different design disciplines within a single assembly [53]. The individual components and functions are linked together in an intricate web of synergistic relationships through which the design becomes more powerful and complex than the sum of the individual pieces.

To create and manage the design of complex products, designers proceed through a series of process phases. Evbuomwan et al. characterize these phases as divergence, transformation, and convergence [20]. The designer first extends the solution space by diverging from the well-known aspects of the design situation while identifying features of the problem that permit a valuable and feasible solution. Creativity, pattern-making, insight, and guesswork allow the designer to transform the results of the divergent search into patterns that may lead to a single design. Eventually, the designer must converge to the final design by removing uncertainties and design alternatives.

Pahl and Beitz present a similar view of the design process in which the more familiar terms of conceptual design, embodiment design, and detailed design are

used to describe the major phases [41]. The conceptual design phase determines the principle solution by abstracting the essential problems, functional structures, and working principles and combining them into a conceptual structure of the design. During the embodiment phase, the designer applies technical and economic knowledge to the development of an overall layout, preliminary component shapes and materials, and production processes. Finally, in the detailed design phase, the arrangement, forms, dimensions, surface properties, materials, and production possibilities are specified, analyzed, and revised into an economical, manufacturable, working product design.

In both of these design process views, designers apply their considerable domain knowledge and experience to the understanding and formation of a product design. Designers must determine customer needs and must have the scientific and engineering knowledge to form these needs into a working product design. The information derived from this design knowledge is highly complex and contains many interdependencies. In addition, the applicable information frequently changes as the design evolves.

Customer needs are characterized by functional and performance requirements that are constrained by the operational environment, budgetary limitations, and other restrictions. Often, customer needs are ambiguous and incomplete and change considerably over time. Designers need to transform these ambiguous requirements into a concrete design model while accommodating any changes. Unfortunately, these ambiguous, changing requirements are a frequent cause of cost overruns and delays in product development.

Engineering design requires considerable knowledge of scientific and engineering principles. In addition, information about existing designs, standard components and materials, and manufacturing capabilities must be available to the designer. This diversity of information and knowledge often requires multiple designers, with expertise in different engineering disciplines, to cooperate in the design of a single product.

The intricate relationships and large quantities of information in a complex

design are very difficult for a design team to assimilate. The team must organize and abstract the design information in different ways to explore various design possibilities, to analyze cross-disciplinary design compatibilities, to organize design ideas into feasible design layouts and patterns, and to revise and restrict the design alternatives until a workable design is obtained. Designers have developed a number of techniques for abstracting design information and managing the complexity of product designs.

One technique is to break the problem into a number of smaller subproblems, each of which is less complex than the original. If done properly, these smaller problems can be resolved simultaneously by separate design teams, then the solution can be integrated together to form the complete product design. In some cases, existing designs may be reused as solutions to design subproblems. In fact, standardized catalog parts and components are frequently reused in this manner.

In the early stages of design, complexity is frequently reduced by deferring specification and modeling of many of the details, both geometric and functional. In these early stages, functional concepts are embodied in high-level components that interact in a specified manner. Multiple alternatives may be developed and analyzed before a particular design approach is selected. As the design problem becomes better understood, the alternatives are narrowed down, additional detail is added, and the design is more rigorously analyzed. This cycle continues until the design has evolved into its final form.

As a design problem is decomposed into subproblems or as detail is added at different levels of abstraction, additional relationships are established between components of the design. These relationships evolve along with the design components. Understanding and ensuring compatibility with these relationships is critical to designing a successful product. This is often complicated, however, by the difficulty in capturing and defining these relationships.

Ideally, one would dedicate sufficient resources to completely identify, specify, and analyze every aspect of a complex design. Since resources are usually limited, however, one can reduce the chance of product failure by concentrating resources

on those areas that pose the greatest risk. The relationships between design components have considerable impact on the overall design due to their synergistic effect. Consequently, these relationships provide a convenient focal point for minimizing design risk.

### 1.2.2 Computer-Aided Design

Computer-aided design (CAD) systems are essential for creating and maintaining complex product design information. Most CAD software tools emphasize detailed modeling of individual design components, but often fail to support the complex relationships between design components that are typical of most actual product designs. As a result, the design team must take additional steps to manage these relationships independent of the actual component models.

Many different CAD software tools have been developed for supporting different phases of the design process or for representing different aspects of the design model. Since the high-level conceptual models, detailed design models, and analysis models are created with different domain tools, the model for a single design component is often maintained in multiple, incompatibly formatted files. The same is true for functional, geometric, manufacturing, and assembly models that are created with different tools. In addition to the different file formats, each tool operates in its own workspace with its own set of commands and procedures.

For a software tool to use models that are created by a different tool, some sort of transformation is required, often involving translation of design formats, manual conversion of design information by the design team, or additional design steps. This frequently results in information lost during the translation, time lost to accomplish the translation, and additional complexity by having to keep track of the mapping between representations. Design data that are shared between tools must be organized in a fashion that is efficient for translation. This often means large pieces of the design model are grouped in a single file. Changes made by one tool are not available in other tools unless the designer takes explicit steps to transform the changes into the proper format. As a consequence, it is difficult to propagate incremental changes between tools. Reuse of a design model is also

made more difficult, since the designer must extract each representation from the different tool workspaces in which they are defined.

In this independent workspace paradigm, links between different component models are difficult to specify. Components that are composed of other independently modeled components have no way of showing these connections except to make copies of the other components. If one of these components changes, a new copy must be inserted into the aggregate representation. If a component must interact with other components, this interaction can be specified independently within each component; however, there is no easy way to determine which other components are compatible with that specification.

Product data management (PDM) tools can be used in conjunction with CAD tools to specify and manage structural relationships between design components. The relationships specified by these high-level tools, however, fail to capture complex design information such as functionality, strength of materials, or geometric constraints. While these tools help the designer determine which components are related, manual intervention is still required to keep the independent design representations consistent. PDM tools are also restricted to managing complete design files, thus limiting their utility for managing incremental changes.

Associated with the various phases, levels of detail, and revisions of a design model are the rationale and decisions that describe how the model evolved to its current state. These informal, text based descriptions are often maintained in a loosely organized set of notes or in the minds of the designers. Although this information is often essential for design corrections or for acquiring knowledge about the design, the information is seldom incorporated into CAD models.

Many components in a mechanical design, such as bolts, bearings, springs, and other connectors, are standardized and produced by independent manufacturers. Since these components are not new designs they are often inadequately represented in detailed design models. Where they are represented, the designer usually has to individually specify features to accommodate these components in each affected part.

## 1.3 The Problem

Whether for conceptual or detailed design, a major shortcoming of most CAD systems is the isolation of the individual design artifacts. Models for individual components, different disciplines, and other design characteristics are developed and maintained in separate files and formats from related representations in the overall product design.

To manage the synergistic relationships that exist between design representations, the designer must currently document and track these relationships independent of the actual component models. Product data management tools attempt to automate this process by providing structural and classification links between detailed design representations; however, existing tools are unable to depict the synergistic information that is essential to these relationships.

The file-based organization of product data management tools and manual filing systems forces designers to manage design representations at a large granularity. A large number of design changes, however, require relatively small changes to a particular aspect of a design representation. Since data management tools can only deal with changes at a large granularity, the designers must intervene manually to ensure consistent changes are made in all related representations. This process is time consuming and increases the chances of introducing errors into the design.

## 1.4 Managing Complexity

The intricate relationships and large quantities of information in a complex design are very difficult for a designer to assimilate in their totality. To understand and manage this complexity, designers must be able to organize and constrain the design to limit the amount of detail or the number of possibilities to be analyzed at any given time. As the design evolves or requirements change, the designer must be able to reorganize and modify the design constraints so that useful solutions are not overlooked.

To automate the management of complex design models, a CAD environment must provide representations and constraints for controlling and analyzing the behavior of interacting design components. These representations must be organized

into a single product model in which the designer can specify and track design models at multiple levels of detail, through multiple revisions and alternatives, and across different design perspectives.

Modern production environments dictate additional capabilities that must be accommodated in the management of design complexity. Geographically separated design teams may work concurrently on different sections of a design or may collaborate by providing unique expertise to a single design. Existing electronic designs may be cataloged and stored in a standard format for reuse by anyone on a computer network. Independently developed tools are available for automatically checking and analyzing different design characteristics. A modern CAD environment should enable these capabilities by seamlessly integrating the different tools and representations with the necessary data and process relationships.

### 1.4.1 Terminology

To facilitate understanding and analysis of design complexity, this research examines three organizing paradigms – aggregation, interaction, and variation – that are inherent in the creation of any complex product design.

- *Aggregation* is the organization of related components into collections and the arrangement of these collections into a product hierarchy. Aggregation depicts the decomposition and abstraction of the design model at multiple levels of detail.

- *Interaction* is the description of how two parts or subassemblies fit together and cooperate to provide new capabilities.

- *Variation* is the evolution of a design as it changes over time.

Although these organizing paradigms are common, the preceding definitions are particular to this work since no standard definitions exist.

### 1.4.2 Objectives

To facilitate complexity management in a modern production environment, this research introduces an automated framework for organizing and controlling complex design models. This framework incorporates the aggregation, interaction, and variation paradigms defined above into a single structure for specifying and organizing complex product designs. Specifically, this framework is intended to assist the designer by:

- Integrating different design disciplines and related design components into a single product model. A single product model simplifies analysis by having information available in a single structure and a similar format. Design changes are more readily propagated through a single design model making it easier to analyze new designs or the impact of changes.

- Supporting independent manipulation of different design representations and components along with the relationships between them. By independently manipulating relationships, designers can communicate between related components and control how components affect one another. Designers are not constrained to a particular level of detail or a particular representation when modifying design components; rather, changes can be made at any level and propagated to related representations. Independent manipulation of design representations also makes it easier for designers to simultaneously work on different parts of the product or to reuse existing product designs.

- Providing flexibility to manage and evolve design representations for different design processes or applications. Designers can organize and evolve their designs according to the process that is most beneficial to their particular situation. Data structures implemented for a particular application such as force analysis or manufacturing process planning, can be readily extended to incorporate additional design disciplines or application areas.

### 1.4.3 Methodology

The primary components of this framework are based on the organizational paradigms defined in Section 1.4.1. These components include *aggregation objects* for organizing related information into collections, *relationship objects* for describing the hierarchical and interaction relationships between components, and *version objects* for recording the evolution of a design as it changes over time. Each of the aggregation objects, relationship objects, and version objects contains automated "software assistants" that assist the designer in analyzing and managing the product model.

An *aggregation object* is an organizational structure that encapsulates multiple design entities into a single design object. Aggregation objects can be nested within other aggregation objects to form a decomposition hierarchy with different levels of detail. An aggregation object creates a scope into which the designer may insert related geometric, functional, manufacturing, or other design information. The scope of the aggregation object restricts access to the encapsulated components from objects external to the aggregation.

This framework contains two type of relationship objects – *attachments* that represent the hierarchical relationships between design components, and *interface specification objects* that represent the peer-to-peer interaction relationships between components. Relationship objects contain design constraints and methods for analyzing and validating related components. Both types of relationship objects are distinct design objects that can be independently manipulated by the designer to analyze and control the related components. The attachment relationship links components that are part of the same higher-level aggregation. Interface specification objects describe the detailed interactions between parts in an assembly aggregation.

To complete the framework, this research creates *version objects* for recording and managing design modifications. When a designer modifies a design aggregation, a new version object is automatically created that computes and records the differences in the design. Version objects can also be created to record alternative

solutions or views of the design. Version objects are derived from aggregation objects so that designers can control granularity by the number of components in an aggregation. Simple commands are associated with the version objects for interactively selecting or copying any existing version of a component model.

### 1.4.4 Design Characteristics

In an attempt to manage design complexity, designers employ a number of design techniques to decompose the problem into more manageable pieces and to control changes to the design as it evolves. For a CAD system to support the entire design process, it must accommodate these complexity management techniques along with the specification of geometry and functionality. A CAD system should also enable the designer to maximize utilization of the storage, communication, and processing power of modern computers systems and networks. This research considers the following design techniques and characteristics essential for representing and managing design complexity in modern CAD systems.

*Decomposition.* A high level concept of the design is decomposed into smaller components that are more easily understood and implemented.

*Simultaneous Development.* Different designers work on separate components of the design at the same time.

*Integration.* Components designed separately from each other are composed into a higher level functional design. Independently designed components should be compatible with the high level specifications.

*Nongeometric Design Representation.* To adequately analyze the feasibility and performance of a design model, the designer must be able to quantify information concerning the functionality, ease of assembly, manufacturing processes, and other design disciplines. This may involve kinematic joints, force constraints, manufacturing or assembly features, fasteners and connectors, or other specialized design components.

*Design Exploration.* A designer often explores multiple alternatives before the design is completed.

*Concurrent Design.* Designers with expertise in different design disciplines may need to concurrently develop and analyze the design model from multiple viewpoints.

*Design Recovery.* Once a design has evolved, a designer may determine that another version is more accurate. This requires recovery of a previous version or alternative of the design.

*Design Reuse.* A design may be adapted and reused to satisfy a different set of requirements.

*Refinement.* Once a design exists, this design may be refined to adapt to different requirements or to improve the ability of the design to satisfy existing requirements.

*Change Management.* A designer may need to propagate a change to interacting design components, alternate versions, or higher level design aggregations so that different representations of a model are kept consistent. A designer may also want to restrict how changes are made and propagated through the model. Before a change is made permanent, a designer may want to analyze the impact that the change has on the remainder of the model.

*Design History.* The designer needs to keep track of design decisions and the history of the design to reduce rework and to allow different individuals to understand how the design has evolved. Design history also assists with corrections to a design by providing an understanding of why a particular design decision was made.

*Assisted Analysis.* Some quantifiable elements of the design can be automatically analyzed. To better assist the designer, the design environment should support such automation.

*Simulation.* The designer may want to simulate the movement and operation of an assembly to analyze interference or behavior.

In addition to these design capabilities, it is important that any design system is easy to use and can be readily extended to support other design applications. Toward this goal, this research strives to support complexity management in a manner that allows the design team to increase its productivity while proceeding according to a process that is comfortable to its members. Designer productivity is enhanced by automating tedious tasks, supporting interactive editing and analysis, and minimizing user interface complexity. In addition, the relationship and aggregation objects are designed with considerable flexibility so that they can be easily extended to represent multiple design disciplines or applications.

### 1.4.5 Alpha_1 Design Environment

The complexity management framework in this research is integrated into *Alpha_1*, an object-oriented testbed system supporting research into geometric modeling, high-quality graphics, curve and surface representations and algorithms, engineering design, analysis, visualization, process planning, and computer-integrated manufacturing [55]. *Alpha_1* provides geometric primitives, surface and curve representations, and mechanical features that can be used with the aggregation, interaction, and variational mechanisms presented in this research to provide a powerful computer-aided design and manufacturing environment.

Mechanical models in *Alpha_1* are represented by a directed graph that identifies the prerequisite objects necessary to construct a particular object and the dependent objects that are based on the object. The model graph is used to propagate changes to dependent objects and to minimize processing by computing only the necessary prerequisite objects.

The *Alpha_1* object-oriented software development environment facilitates code generation for new modeling objects and provides a standard framework for building model object constructors to integrate model objects into graphical and textual user interfaces. The controlled interaction and aggregation mechanisms are implemented

as independent *Alpha_1* model objects that can be manipulated and controlled like any other model object in the system.

### 1.4.6 Limitations

Although a powerful user interface is a vital component of any design system, this research does not specifically address user interface issues. Instead, the complexity management mechanisms are implemented as fundamental design objects that can be flexibly integrated into a number of customizable user interfaces in *Alpha_1*.

This research implements versioning mechanisms to support the evolution of a design model as it is decomposed and refined over time. Some fundamental versioning capabilities, available in commercial object-oriented database management systems [30, 65], are implemented as a basis for the complexity management capabilities introduced in this research. These fundamental capabilities are extended to support the management of alternative solutions and views, user-controlled granularity, and the use of versioning as an interactive design tool.

## 1.5 Document Summary

There are many aspects of automating the management of design complexity that have been previously explored by other researchers. This work is described and analyzed in Chapter 2.

To provide a better understanding of the automated mechanisms and to demonstrate the capabilities of the automated framework that is introduced in this research, two case studies were undertaken with real manufacturing design examples. These case studies, along with design methodologies for simultaneous and incremental design, are outlined in Chapter 3. Examples from the two case studies are interspersed throughout the remainder of the document.

Chapter's 4, 5, and 6 describe the fundamental mechanisms and strategies for the automated complexity management framework introduced by this research. Chapter 4 explores the roles and data structures associated with part, assembly, and other aggregations. Chapter 5 describes the complexity associated with the interaction between parts in an assembly along with the data structures used

for the specification of this interaction. Chapter 6 presents the data structures and capabilities of the versioning mechanism that is used to maintain revised and alternate variations of a design aggregation.

In Chapter 7, the results of this research are measured against the capabilities and characteristics identified in Section 1.4.4. These capabilities are also used as a basis of comparison for other design data models. Chapter 8 concludes this document with a summary of the research and conclusions about the contributions of this research to the field of computer-aided design. This chapter also recommends future research directions.

# CHAPTER 2

# RELATED WORK

Complex product design is characterized by a variety of interrelated process activities, design representations, and model components that evolve over time. As discussed in this chapter, however, most existing tools and research support only static representations or component models with minimal support for the relationships between these representations, the variations as the models evolve over time, or the design activities that are necessary to manage design complexity.

Feature-based design is a common approach for embedding different functions and multiple design disciplines into a single part model. A feature is a standard, reusable design entity that encapsulates related functional, manufacturing, geometric, or other engineering information into a single representational abstraction. A sampling of these feature-based design approaches is described in Section 2.1.

Some researchers have developed data models for design that incorporate fragments of information associated with the relationships between design components in a complex design. These models, as summarized in Section 2.2, range from data structures that integrate structural and constraint relationships into the design model to mechanisms for simplifying the specification of some of the relationships that contribute to design complexity.

As a design evolves over time, a number of model variations are created. These variations are supported with version management capabilities as discussed in Section 2.3. Unfortunately, version management capabilities are not well supported in CAD systems.

Product data management (PDM) systems take a different approach to managing complexity as described in Section 2.4. Instead of embedding complexity information in the original CAD models, PDM systems maintain a separate database

that links together the individual component models created by different CAD applications.

## 2.1 Feature-Based Design

A common technique for controlling design complexity is to hide some of the details at different levels of abstraction. Feature-based design facilitates this approach by encapsulating geometry, functionality, design intent, tolerances, manufacturing processes, or other important design information into reusable, standardized features. Features help control complexity by enabling the designer to work with a single entity instead of many separate pieces of information. Specialized features may be developed for different design disciplines, enabling concurrent design by multiple designers working with different feature views.

Shah and Mäntylä [49] characterize a feature as a physical constituent of a part that has engineering significance and predictable properties and is mappable to a generic shape. Recurring characteristics of products may be modeled as feature classes that can be reused to facilitate construction of a product design. Features provide a means for "chunking" information, making it easier for humans to understand. According to Shah and Mäntylä, "a major advantage of features is that they provide an additional level of information to CAD systems to make them more useful for design and to integrate a design with downstream applications. Because of the higher semantic level of features, they can provide a basis for recording a more complete product definition."

Features are commonly used to represent manufacturing processes associated with a particular shape of a part (for example, the drilling or reaming processes required to machine a hole or a pocket) [7, 9, 10, 24, 54]. By embedding process information in the feature, a process plan to manufacture the part can be generated automatically [10]. Tolerance and dimension information is also frequently encapsulated in features, providing a convenient mechanism for automatic analysis of associated costs and ease of manufacturing [26, 44, 60]. Many researchers discuss embedding functional requirements and specifications within features for design ver-

ification [2, 9, 22]. Implementation of functional feature modeling systems, however, has been limited to very small, research domains, probably because of the difficulty associated with unambiguously specifying functionality. Features have also been used to represent assembly relationships and constraints [16, 36, 49, 50, 59].

A feature-based design system must consider location of a feature on a part and relationships between features on a part or in an assembly. Feature validation and interactions between features on a part are also important issues. If multiple feature views exist that represent the same design component from multiple perspectives (for example, a functional view or a manufacturing view composed of the appropriate functional or manufacturing features), one must be able to map between the different views. In addition, to represent information such as manufacturing processes, specific features must be defined to describe those processes. These feature-based design issues are dealt with in a variety of ways as discussed in the following paragraphs.

Location of a feature on a part is usually determined relative to some geometric entity (for example, a face or an edge), to another feature, or to a user-defined reference. In the University of Utah's *Alpha_1* system [10, 55], a designer defines an anchor to specify location and orientation. Each feature also has an anchor and the feature is placed in the model by aligning the feature anchor with the user-defined anchor. Process plans built from features in *Alpha_1* have successfully produced a wide variety of machined parts; however, placement of features requires the designer to ensure that anchors are properly specified. Ranyak and Fridshal [44] resolve planar and cylindrical geometrical features into primitives (point, line, or plane) and locate the feature relative to another feature or a datum reference frame. Location tolerances are embedded in the feature to determine the type of location constraint to apply (for example, distance, concentricity, angle). Gossard et al. [26] use location dimensions to locate a feature relative to a particular face. Relative position operators for specifying the intersection angle of two faces or the distance between two parallel planes allow the designer to define and locate features with scalar values.

In an assembly, feature relationships may be used to constrain how parts fit together. Driskill [16] defines assembly features such as a peg in a hole that constrain the geometry and the relative location of the peg and the hole so that the two parts fit together. In her work, the peg and the hole are separate features that act together to form an assembly. Shah and Tadepalli [50] present another approach in which a new feature is created, in addition to the peg and the hole on each part, that does not fit on any part, but describes and constrains the relationship between the features on each of the two parts. This approach is used by Shah and Tadepalli to determine if two parts can be assembled. In both of these approaches, individual parts are designed independently and include one component compatible with an assembly feature. Once designed, these parts are selected and analyzed to determine if and how they can be assembled. If no valid assembly representation is possible, the parts must be independently modified and reanalyzed until a valid configuration is reached. Both approaches are also limited to static assemblies. Assembly features, as defined by Driskill or by Shah and Tadepalli, identify compatible geometry and mating constraints that are useful for determining whether two parts may be assembled, but are not intended for specifying or controlling the interaction of the parts once assembled.

By properly specifying feature constraints, features become valuable tools for validating the geometry or other attributes of a model. For example, a through hole can be specified so that its entire diameter is on the part and its depth is equal to the thickness of the stock. Any time the model is changed, features can be revalidated to make sure all constraints are satisfied. Unfortunately, this problem is easily complicated by interactions among features. For instance, if one of two parallel slot features is widened such that it intersects with the other, the two separate features have changed into a single slot feature.

Geelink et al. [24] group interacting features into a compound feature that can be decomposed into its primitive features for process planning. Unfortunately, intersection of two or more features may cause deletion of important portions of the geometry. Geelink et al. define feature recognition algorithms that alleviate

this intersection problem by relaxing feature definitions. This solution has been implemented for a limited set of features, but it is not apparent that the solution is generally applicable to other features or to all feature configurations. When feature constraints are violated by a change, Dohmen [13] reconstructs the feature model from the geometric primitives. This requires that all features and constraints be programmatically defined with low-level geometry. Chen [8] matches feature vertices, edges, and faces to determine when a feature is no longer valid. He then rebuilds the feature model by deleting or modifying invalid features. Because of the ambiguity in determining how features should interact, Chen's approach often results in an approximation of the feature model. All of these methods require interaction with the low-level part geometry, degrading the higher-level abstraction provided by features.

Feature modeling is frequently proposed as a method for concurrent design. Many different aspects of the design, such as functionality, manufacturing, and assembly, are considered concurrently to accelerate the design process. To analyze each of these design aspects, different design views are needed. To analyze functional capabilities, the designer needs to look at functional relationships and constraints. A process plan for manufacturing must be generated and analyzed for efficiency and cost-effectiveness. By providing features to represent each of these views, and mapping between the feature views, designers and analysts with different expertise can analyze the model at the same time. Unfortunately, mapping between views and keeping them consistent is a considerably difficult task.

Shah [49] identifies four theoretical approaches to feature mapping. Heuristic methods use prespecified transformation rules to map between two engineering application views. Another approach transforms features to an intermediate-level structure that is common to multiple applications. Cell-based mapping decomposes features into cells that can then be transformed into another feature view. In graph-based mapping, feature attributes and constraints are represented by a graph that is transformed, using graph grammars and algorithms, into another graph forming a different engineering perspective.

Falcidieno et al. [21] extract shape information from a feature by applying previously defined feature and shape rules. The shape information is stored as a hierarchical graph that can be converted to different views using predefined routines. All features and views, including feature interactions, must be explicitly defined by an application expert. It is not clear how robust or complicated this process is, but the examples have less than 20 shape features and are limited to planar and cylindrical faces. Brooks and Greenway [7] use object relationships to relate different feature views to the faces and topology of the geometric model. This work requires programmatic definition of features and is limited to planar and quadric surfaces. Cunningham and Dixon [11] provide a mechanism for defining heuristics to transform between a design feature and any alternate activity representation. A monitor routine restricts the combinations of design features to those that can be converted into activity features. All features and their mappings to alternate activities must be explicitly defined before the monitor routine will allow them to be used in the design. Intersections of more than two features are derived from adjoining two-way relationships. Wearring [61] identifies intermediate geometry features that can be reorganized, through detailed geometry manipulation by the designer, into whatever functional feature is desired. For a simple block with a hole in it, the relationships, dimensions and tolerances for three of the faces and the hole must be specified and maintained by the designer. In practice, each of these implementations has only been applied to a limited domain and to parts with limited complexity. The solution space and complexity for many parts or assemblies quickly become unmanageable.

A significant drawback to any feature modeling system is the domain specific nature of features. To model a different manufacturing domain or a different view, a new set of features is required. Some researchers have tried to overcome this with interactive feature definition; however, due to the difficulty in specifying relationships and constraints, only limited analysis and validation is possible in these systems. The FROOM (Feature and Relation based Object Oriented Modeling) system [24], for example, supports only planar, cylindrical, and conical faces with adjacent,

perpendicular, parallel, and coaxial relations. Only features that can be completely defined with parameters are allowed. Other researchers [44, 47] have developed an object-oriented feature hierarchy, where new features inherit attributes and constraints from parent feature classes. Transformation and recognition of these features is based on the predefined, high-level parent class and may not consider the necessary detail contained in the feature object. Features that do not fall into a preexisting class still require new class definitions.

Researchers attribute considerable representation and modeling power to the use of features. In practice, however, the only common use of features, other than for representing geometric attributes such as dimensioning, tolerancing, and shape, is for manufacturing process automation. Very little functional specification and analysis is supported by existing feature modeling systems. Features are also very application dependent and mapping between feature domains is complicated by the interactions between multiple features on a part. Representing and mapping between complex parts with multiple interacting features is difficult to do with existing systems.

## 2.2 Data Models for Design

Even though features facilitate representation of different functions and design disciplines in a design model, they are independent design objects and contribute little to the organization of the different features into manufacturable parts or assemblies. A number of more comprehensive models have been proposed for linking features and other design information together into complex parts and assemblies and embedding these relationships into a static product model. Some of these models utilize structural and constraint relationships to integrate individual component models into complex aggregations whereas others simply facilitate specification of complex relationships. Some models focus on high-level concepts and functionality while others emphasize detailed manufacturing designs.

Eastman and Fereshetian [19] present a set of criteria for evaluating and comparing product data models. These criteria include an object-oriented class hierarchy

with abstract data types, multiple specializations of classes, and composite objects. A data model must support relations within composite objects, relations between variables, and relations such as cardinality and dependency between object structures. Included in these relations are constraints and aggregations. Both invariant and variant relations are required. Relations must support integrity management of the data to include partial integrity while the design is in an intermediate state. In addition, the product data model must provide for continuous object refinement and schema evolution.

Eastman's *Engineering Data Model (EDM)* [17, 18] for architectural design is among the most comprehensive of the design models reviewed in this research. EDM is an architectural design model that strives to represent function and form at multiple levels of abstraction with explicit management of partial integrity. EDM provides aggregation, composition, and accumulation relationships that allow the designer to describe the aggregation hierarchy of the design along with constraint information between components. EDM is based on set theory and first order logic. Domains are sets of values corresponding to a simple type, aggregations are sets of named domains, and constraints are general relations stored as procedures. The primary object is a *functional entity* – an aggregation and its constraints along with other entities that it specializes. A *composition* is the set of relations linking an entity to its parts. These relations are defined as *accumulations* that include functional design rules and property relations between the parts. To support partial integrity, some relations are not satisfied immediately. Integrity between multiple views is maintained through *maps* that are specializations of constraints that can change the database variables and schema. Missing from EDM are operations, such as automated generation of relations, that simplify designer interaction and explicit version management of design revisions. A number of architectural design domains, including composite windows, core and panel walls, and basic building structures, have been modeled with EDM. Due to its architectural focus on static structures, however, it is not clear that the relationships in EDM can incorporate mechanical interaction information such as forces, connectors, and kinematics. Manufacturing

features and other mechanical design representations have not been demonstrated with EDM.

Gui and Mäntylä's *multigraph* structure [27] focuses on the top-down evolution of an assembly design from high-level functional concepts. The multigraph supports multiple levels of detail and provides links between functional, structural, and geometric information. A leaf node in the multigraph can be linked with a functional description, geometry, features, elements in a bond graph, or other design information. The multigraph also provides a *connector* for describing force transmission and motion constraints associated with the interaction between parts in an assembly. An example connector is a spring that imparts a force but also provides a physical geometric connection. A *feature link* relates design functionality or other feature representations to the geometry. Gui and Mäntylä use this multigraph representation to share design objects between three system components – the *DesignPlanner* that describes functional relationships; the *DesignSketcher* supporting geometric modeling; and the *DesignConsultant* that resembles an expert system. Each system component links its design representation to the object multigraph. Once the designer has specified the functionality, tools for behavior and energy transformation analysis can be applied to the multigraph. The designer develops geometric representations and associates them with the proper functions. Gui and Mäntylä describe how the multigraph representation and associated design and analysis tools are used to model an electrical contactor used to open a circuit based on a control voltage. Designers are given considerable flexibility in representing functionality; however, this flexibility limits the degree to which the analysis is automated. The multigraph emphasizes functionality and assembly modeling, but requires that the detailed manufacturing information be modeled separately. While linkages exist, the multigraph mechanisms are not applied directly to the specification and validation of individual, manufacturable parts.

Representing design functionality is a common goal of many researchers. The feature-based approach discussed in Section 2.1 is intended to support functional representation, but has rarely been used in this fashion. Baxter et al. [2] propose an

enhanced entity-relation diagram for representing design functionality and analyzing how well a product satisfies the specified functionality. Functional relationships such as *performed_by*, *input_of*, *output_of*, and *has_need_of* are traversed and the functionality of the linked components is analyzed to determine if these relations are satisfied. Baxter et al. tested their model on a valve assembly with 22 components. The model contained 35 function instances and approximately 1000 nodes. It is not clear how much of the validation was automated; however, some human intervention appears necessary to resolve the ambiguity associated with integrating subfunctions and analyzing their combined ability to perform their parent function.

Rosenman and Gero [45] assert that multiple views and representations are dependent on a functional context. Different views (for example, architectural, mechanical, and structural) are composed of a different set of functional primitives rather than a different look at the same standardized primitives. This requires a different model for each view with a view defined by a set of functions or a set of functional systems. Different disciplines may refer to the same element using different terminology. This is handled using explicit relationships between elements with identical properties, elements in an assembly, partial elements, and constrained elements. This data model is used to create architectural, mechanical, and structural views of a building; however, change propagation and other relationships between the views are not demonstrated.

Gorti and Sriram [25] present a framework for conceptual design that uses functional, composition, aggregation, and spatial relationships. The designer selects predefined components, establishes functional relationships between the components (supports, transmits load, or resists load), and specifies the spatial relationship (for example, connects, intersects, or abuts). These relationships are used to generate possible design concepts to use as a basis for more detailed design. A limited set of conceptual entities such as pier, slab, and bank, have been developed to demonstrate this approach for the design of a river bridge.

The interaction between components in an assembly is inadequately represented in many data models. Models such as Eastman's EDM include hierarchical aggre-

gations and positioning constraints, but provide only limited support for describing how different aggregations or individual components interact. Other models such as Driskill's assembly features, incorporate interaction information into individual parts which restricts analysis of the interaction relationship and complicates change propagation between the interacting parts. Some researchers, however, have realized this problem and have focused on the specification and analysis of the interaction relationships between parts in an assembly.

Bordegoni and Cugini [5] specifically address the interaction between fixed components in a mechanical assembly. They propose an *assembly feature* for specifying the interaction relationship at various levels of detail. This is accomplished by having the designer fill in appropriate detail information in a cataloged template for each instance of an interaction relationship; however, if the template does not provide a slot for the information, the detail can be added only after modifying the template. Multidisciplinary analysis is facilitated by providing functional, positioning, and assembly information in a single relationship. Bordegoni and Cugini's implementation of assembly features, however, is rather limited, having only been demonstrated for fixed assemblies with no kinematic interaction.

Lee et al. [34, 37, 38] developed mating features to represent four typical positioning and kinematic configurations between planar and cylindrical surfaces of parts in an assembly model. The *against* mating feature specifies that the surface of one part must lie against a second part. This relationship has one rotational degree of freedom and two translational degrees of freedom. A *fits* mating feature specifies a cylinder in a hole. Here translational movement is allowed along the axis of the cylinder and rotational movement is allowed around the axis. A *contact* feature is an *against* feature with no movement and a *tight fit* feature is a *fits* feature with no movement. The designer associates mating features with individual part surfaces and, if a valid set of mating features is specified, the modeling system generates the necessary equations to infer the relative position of the parts.

Beach and Anderson [3] extend the mating feature concept to include a total of twelve different attachments. Their attachment hierarchy includes cylindrical,

planar, revolute, prismatic, spherical, and helical attachments that are specified as either rigid or a kinematic pair. They represent these attachments in a general graph showing the attachment relationships a part has with all other parts. This graph is supplemented with a hierarchical tree to show subassembly grouping. All attachments within a subassembly must be rigid. If a component is modified and does not violate any constraints, the parts are automatically reassembled. A simple wheel mount, with no subassemblies and with only planar and cylindrical attachments, is provided as an example. The subassembly hierarchy, although critical to reducing the model complexity and increasing designer understanding, is included by Beach and Anderson almost as an afterthought. The subassembly hierarchy is implemented in a separate data structure and there are no relationships between this hierarchy and the attachment graph. The two structures are integrated only through high-level software routines. Use of the subassembly structure is not illustrated in any examples.

Wolter and Chandrasekaran [63] use geometric structures, called *geomes*, to represent "any arbitrary collection of geometric elements whose form may or may not be fully specified." Geomes can be used for relationships between objects as well as the objects themselves. Functional information can also be associated with a geome. For example, the designer can specify the behavior and a limited amount of geometry (such as the axis of rotation) for a kinematic constraint. The constraint can then be instantiated anywhere in the model by specifying the necessary parameters. Higher-level geomes can be used as design specifications with the implementation represented in lower-level geomes. Geomes can also be used to represent geometric entities that have no physical existence, such as the axis of a hole or the paper path of a copying machine. Wolter and Chandrasekaran provide a simple example of a device that uses two *rack-and-pinion* geomes to transform translational motion in one direction into translational motion in a perpendicular direction. While their approach is quite flexible, Wolter and Chandrasekaran point out that a product designed with this approach is certain to be more complex than a geometrical representation alone since a considerable amount of nongeometric

constraints may also be included in the geomes; however, by organizing the data hierarchically, the amount of information presented to the designer can be limited, thereby facilitating understanding and manipulation of the design. Wolter and Chandrasekaran also discuss the difficulty of graphically representing this information and in unambiguously interpreting the constraints. Since much of the framework presented by Wolter and Chandrasekaran has not been completely implemented, the geome concept has only been demonstrated for hypothetical examples. It appears to be highly flexible, however, and one can easily envision geomes as an interface specification between two objects in an assembly, as relations between views, or as constraints imposed on an aggregation hierarchy or different design alternatives.

Fasteners and connectors are often critical to the interaction between parts. Salomons et al. [46] propose a mechanism for incorporating connection information such as a weld or keyway into a relationship describing the interaction between parts. These relations do not appear to be used for any automated analysis or validation. Abrantes and Hill [1] incorporate fasteners into a relationship between assembly parts; however, their fasteners are used only as a means for reducing the number of possible assembly configurations.

A key aspect of any design is the evolution of the design model over time as it proceeds through the design process. Some researchers have proposed representations for recording the historical information associated with this evolution. These representations enable the designer to embed historical information directly in the design model.

Kim and Szykman [33] use design decisions to describe the relationships between versions of a design model. The concept behind their approach is that any time a design change is made, it reflects a decision by the designer. Design decision relationships facilitate the representation and exploration of design alternatives. By forcing designers to document design decisions, versions are more easily associated with new functionality or abstractions rather than simply representing a snapshot of the design at a particular point in time. A conceptual example of a television re-

mote control is presented in which different battery configurations are interactively examined and analyzed. Considerable flexibility is provided for representing design knowledge with these relationships; however, these flexible representations limit the amount of analysis and constraint checking that can be automated. Design decision relationships are static objects and must be explicitly defined by the designer.

Shah et al. [48] classify design history information into four conceptual elements: the design problem, domain knowledge, design processes, and the design solution or product data. Each of these elements must be captured in representational data structures to form a design history data model. Shah et al. develop a design language to represent the processes, organizational entities, design products, and relationships between the entities associated with a design project at any particular point in time. A number of issues for representing design history, however, remain unresolved. Among these issues are the extension of database technology to incorporate modeling of processes, rationale, and design constraints and the development of a dynamic data definition language that can specify design history and represent the evolution of the design process.

To assist with change propagation and constraint analysis, researchers have developed active relationship objects that execute preexisting procedures when triggered by another object or event. Active relationships localize constraint and change propagation, thereby reducing complexity and facilitating analysis and interaction.

Sullivan [52] depicts active relationships with *mediators* that represent behavioral relationships between two objects. A behavioral relationship reflects the behavior one object should exhibit when another object completes an operation or changes an attribute. A *behavioral type* object raises an event when a particular action occurs in that object. The mediator recognizes this event and executes the appropriate action on related objects.

Mediator objects can be inherited and decomposed just like other objects. Aggregation and interaction relations, transformation between representations, and evolutionary mapping between versions can all be implemented with mediators. Sullivan clarifies that behavioral relationships embedded in mediators must be

independent of each other; otherwise a specific ordering of mediator invocation is required. Also, the current implementation only handles binary relationships.

While he advocates the use of mediators during the design process, Sullivan provides examples that are more appropriate for an operational environment. Mediators were developed for use in software, and Sullivan uses them to communicate between application objects such as user interface windows. Sullivan proposes mediators for mapping between design views and versions, however no examples are provided. This mapping would be an appropriate use of mediators, but requires specification of the complex behavioral relationships involved in a design environment.

Brett et al. [6] define a *propagation* as an object similar to a mediator, but limited to *nonancestral* relationships between design objects. A nonancestral relationship is one that "is not already in a parent-child relationship within an object-oriented hierarchy." A propagation "can be conceived as an independent, third-party object which causes mediating software to fire whenever changes to one object must trigger changes to other objects so as to maintain data consistency." The idea behind propagations is to encode constraints within the relationship object or provide methods for accessing a constraint database, then act on those constraints to propagate changes between related objects.

One can imagine using propagations to constrain the interaction between parts or to ensure consistency between different alternatives or views. Brett et al. explain, however, that they have been able to implement only single view relationships between features on a single part. Constraints are hard coded in the object definition and can only be used to represent geometric relationships.

Heinrich and Juengst [28] take a completely different approach in analyzing the connection between components in a technical system. They base their work on "the principle that systems and components interact mainly through interfaces which can be thought of as resources and that the resources demanded and the resources supplied by components have to be balanced." An assembly of mechanical components can be modeled by representing the fasteners and mating features as

resources that are consumed by one part and produced by another. Describing the resources consumed and produced by the environment provides a system specification. Similarly, the problem can be decomposed by describing the resources for subassemblies or individual parts. Heinrich and Juengst have tested their approach with prototypes in a variety of electronic and mechanical applications.

Although Heinrich and Juengst do not propose it, the interfaces through which resources are exchanged could be implemented as relationship objects between components. His resource management approach, however, is more applicable to a system of products in an environment rather than individual products. In fact, Heinrich and Juengst clarify that their approach fails if the function depends decisively on how the components are connected.

As evidenced by many of the data models presented in this section, representing functionality is a significant problem in any modeling system. Predefined relationships may adequately specify some functionality, but are generally unable to capture the complete functionality of a product. Complete functional specifications invariably involve some ambiguity that requires human interpretation.

Relationships objects have been demonstrated as a useful mechanism for representing the interaction between components of a design. Relationship objects can be used to represent a multitude of design information, but most implementations have been limited to single aspects of a design in a limited capacity. Combining relations for different design information would simplify the design interface while providing more capability for analysis.

None of the data models presented in this section completely captures the manufacturing design process. Some are concerned only with functionality or assembly joints. Explicit support for version management is minimal or nonexistent. Data models only capture the static representation of the data and do not deal with automating designer manipulation of the data. Applying these design models to real-world manufacturing problems has found only limited success due to the representational complexity involved.

## 2.3 Version Management

As a design evolves over time, many revisions, variants, or alternates of the design may be created. Version management tracks these differences and ensures that related versions of the design model are kept consistent. While version management has been successfully utilized in some engineering disciplines, especially software engineering, there have been few mechanical design systems that support comprehensive version management of design models.

Mechanical design models have certain characteristics that make version management more difficult. Katz [31] identifies the following characteristics of design data that must be considered when developing a version management system:

- Design data are organized hierarchically.
- Design data evolve. Versions must be able to represent revised and alternative designs for an object. Configurations of versions representing complete design models also evolve.
- There are multiple equivalent or corresponding representations of a design object.
- Design object instances are derived from object classes and inherit attributes and behavior from the parent class.

To support these characteristics, Katz proposes a conceptual versioning model in which the following data primitives are used to describe the relations between design objects:

- Component hierarchies are indicated by `IS-A-PART-OF` relations that form a directed acyclic graph. *Primitive* objects form the leaves of the graph and all other nodes are *composite* objects. This kind of relation is also referred to as an *aggregation*.
- *Version history* is depicted by `IS-DERIVED-FROM` relations that show how one

version is derived from another. Alternatives are shown by multiple parallel derivations of a single version. `IS-A-KIND-OF` relations describe *instances* of a class of objects. Version history and instance relations are depicted graphically as a tree.

- When component hierarchy and history or instance relationships are combined, the result is a *configuration.*

- Equivalent or corresponding objects with different representations are linked via `IS-EQUIVALENT-TO` relations or *equivalences.*

Katz also identifies the following operations necessary for version management:

- *Currency* operations designate and locate the current version of an object or configuration. The current version is the basis for subsequent derivations or equivalent representations.

- *Change propagation* involves automatically incorporating new versions into configurations. *Constraint propagation* refers to the enforcement of equivalence constraints by procedurally regenerating new versions (generating equivalent views). If multiple design objects can have the same parent, propagating changes throughout the design hierarchy can generate an exponential number of propagation paths. Katz suggests that this ambiguity can be minimized by having the designer restrict the propagation paths or by specifying constraints that isolate the changes to a particular part of the design hierarchy.

- *Dynamic configurations*, in which the components in the configuration are not resolved until the aggregation relations are actually traversed, imply methods for describing valid versions to include in the configuration. Dynamic configurations are implemented with various version naming and organization techniques.

- *Workspaces* support simultaneous access to design data. Managing the move-

ment of objects between these workspaces allows designers to make changes to an object without interfering with other designers. Workspaces are typically organized as a master workspace with verified design data and multiple individual workspaces for individual designers. Additional workspaces may also exist for integrating the changes of an entire group of designers.

This framework provided by Katz encompasses much of the existing work in version management and data modeling for design. Katz presents a survey of version management research and describes how this research fits into his framework. Additional variations and implementations of this framework are described in the remainder of this section.

For software configuration management, Zeller [66] presents the concept of *version sets* grouped according to *feature logics.* A feature is a name and a value associated with some element of a configuration item (an example [name, value] pair is [compiler, gcc]). Features may be assigned by the designer or may be derived from the context in which the component is used. By making the delta between two versions a feature, the latest complete version is derived from the unification of all the delta features for that component. Different views can be built from different features associated with the component. System configurations are created by taking the intersection of all relevant features of the components.

In [67], Zeller also discusses four version management models for software that his version sets and feature logics support. The *checkin/checkout repository model* consists of revisions stored in a repository. Designers checkout a component, make changes, and check the component back into the repository. Revisions are represented by *delta features* describing the differences between versions. The *composition model* builds consistent configurations by selecting valid component versions based on features. In the *long transaction model*, a subset of the original configuration is copied to a private workspace. Changes made in the workspace are identified with a feature. When committed back to the original environment, the feature is modified appropriately. The *change set model* allows a change to be integrated

into multiple related components based on selected features. Zeller focuses on building consistent product configurations as sets. While he presents techniques for composing systems and managing version histories, the set representation does not directly support hierarchical aggregation or variational derivation relationships.

Plaice and Wadge [43] present another approach for organizing software versions to implement various configurations of a design. In their work, globally unique names and an ordered version derivation path identify the appropriate version of a component for a configuration. The proper version is identified by matching an extended name (similar to Zeller's features) or selecting the latest version on a similar derivation path. This approach also supports the merging of multiple configurations into one. Holsheimer [29] examines version ordering by decomposing complex logical programming objects into partially ordered sets based on type relations. This ordering facilitates mapping complex objects to a relational database. Each type relation becomes a database relation linking objects according to their object type hierarchy. Version ordering with this approach has been demonstrated for single inheritance and nonvariational object instances.

Schema evolution is a concern of any data model for version management. Meyer [39] states that schema evolution occurs "if at least one class used by a retrieving system differs from its counterpart in the storing system." This causes object mismatches that occur when the schema (or class) for an object has been modified, but the data reflects a different schema. Object mismatches are detected by registering a unique version name or storing a class descriptor with each version. Correction for a removed attribute requires no action, but a new attribute requires some sort of initialization.

Zhou et al. [68] present a far more comprehensive framework for schema evolution in a real-time machine tool control application. Zhou et al. identify schema change taxonomy, schema change invariants, schema change rules, and schema change semantics. The change taxonomy determines schema changes that are significant to the application being supported. Invariants identify those aspects of the schema that must remain unchanged to guarantee database consistency. Examples include

the class hierarchy and distinct object and variable names. Change rules outline heuristics to follow to eliminate ambiguity in resolving schema changes. Specific axioms for resolving the impact of changes on the remaining schema and the underlying data are defined in the change semantics. Zhou et al. have implemented objects for timing constraints and performance polymorphism (different implementations carrying out the same task with different performance measures), but has not yet incorporated schema evolution into the real-time database.

In many instances, the version management system must be able to handle data from different applications. Krishnamurthy and Law [35] implement *meta-operators* (insert, delete, and replace) that summarize all changes made to a version during an editing session. A *compress* operation determines the equivalent meta-operation for a sequence of design tool changes. The meta-operation is then applied to the active version to integrate the changes into the version database. The meta-operations are applicable to any data, regardless of the application or view that created the data. These methods are implemented with a commercial CAD system and demonstrated on a simple shaft assembly. Version representations with meta-operators have not been used with alternate views or applications.

Much of the work in version management concentrates on software development. Although the problems are similar to that in mechanical design, most of the widely used version management systems are based on textual objects. Textual objects are compared word by word for differences between versions. This approach is not appropriate for mechanical design, since such textual comparisons would fail to capture the structure associated with design objects.

In addition to the configuration, organization, schema evolution, and multiple application problems discussed above, there are a number of other issues that must be handled in a version management system. The granularity at which objects are versioned must be determined, either by default or through designer specification. Change propagation can diverge into multiple paths; this must be controlled algorithmically or through designer intervention. Solutions to these issues often require designer interaction, another issue that must be addressed.

Version management is a diverse area with many issues affecting design complexity and evolution.

## 2.4 Product Data Management

Product data management (PDM) systems "integrate and manage processes, applications, and information that define products across multiple systems and media."[42] PDM is a meta-tool, external to any particular CAD application or model, that manages all information used to define a product as well as the processes used to develop those products. Although many PDM products are available commercially, most literature in this area discusses theoretical concepts associated with PDM rather than actual PDM implementations.

*CIMdata*, an international consulting firm focused on PDM and related computer integrated manufacturing (CIM) technologies, classifies PDM capabilities into five functional areas [40, 42]:

1. *Data vault and document management* provide secure storage and retrieval of product definition information. Only authorized users may access data and changes are released only after completing a predefined approval process. Design data are managed as complete documents, images, or files [4] and are frequently stored in a relational database management system.

2. *Workflow and process management* enables the PDM system to control and manage the flow of data between people and applications in accordance with an organization's predefined business processes. Newly completed or modified documents can be automatically routed and tracked throughout the organization for approval and release.

3. *Product structure management* facilitates the creation and management of product configurations. Users can link product definition data such as drawings, documents, and process plans to parts and product structures. Unique views of product information can be configured for different design disciplines. As configurations change over time, the PDM system can track versions and

design variations.

4. *Classification* functions provide efficient mechanisms for indexing and retrieving standard or similar components. These functions have been ignored by most PDM vendors.

5. *Project management* provides work breakdown structures and allows resource scheduling and project tracking. Resources are combined with managed data to provide an additional level of planning and tracking. Project management capabilities are not well supported in current PDM systems; instead, these capabilities are typically provided by third-party project management tools in which a limited number of links are established to the PDM data.

Bilgiç and Rock expand the CIMdata capabilities to include *impact analysis* in which the PDM system detects the effects of a potential design change to the overall product design [4]. In addition to the functional capabilities, CIMdata also identifies utility functions that are provided by PDM systems for communicating between applications and personnel, for transporting data among distributed locations, for translating data between applications, for scanning and viewing images, and for configuring and monitoring the PDM system [42].

In his description of *CIM Manager* [62], Westfechtel provides a more detailed look at some of the issues associated with product and process management for engineering design applications. CIM Manager is conceptual PDM infrastructure for which Westfechtel has implemented a limited prototype. CIM Manager, as well as most other PDM systems, uses a course-grained management scheme in which complete documents are managed. In doing so, it is not possible to manage the individual components or parts that are embedded inside the documents. Westfechtel claims, however, that CIM Manager provides a framework for embedding domain specific tools that can operate on the fine-grained level.

CIM Manager handles relationships between components in the same discipline such as between design representations of components in an assembly. It also handles relationships that cross disciplines such as those between geometric designs

and manufacturing plans. CIM Manager controls versions of individual products as well as versions of product configurations. To deal with these different relationships and versions, Westfechtel identifies three types of consistency control that can be tracked with CIM Manager:

1. *Internal consistency* requires that a design document is consistent with the design language of the tool that created it.

2. *External consistency* means a dependent component or version is consistent with respect to a master. An example of this dependency is a manufacturing plan that is based on a specific geometric representation.

3. *Configuration consistency* requires that a version is consistent with respect to a configuration of components of which it is a member; thus, the version must be internally consistent and externally consistent with all related components in the configuration.

CIM Manager uses a product-centered process management paradigm in which each component of a configuration corresponds to a process that is used to produce the component. Dependencies between components are mapped onto data flows between component processes. CIM Manager supports concurrent engineering by prereleasing intermediate results to dependent processes as soon as possible.

Hamer and Lepoeter [58] describe a more general conceptual framework for managing design data. Their framework is characterized by five orthogonal dimensions. The *version dimension* represents new versions of a model that are modifications of other versions. The *views dimension* accommodates representations at different levels of detail. This may involve different levels of abstraction such as a conceptual view and a detailed view or it may involve different disciplines such as a manufacturing view or a functional view. The *hierarchy dimension* depicts the decomposition of a design model into smaller parts. The *status dimension* corresponds to organizational procedures used to maximize the likelihood that a design is satisfactory. A different status may require a different workspace. Finally,

the *variants dimension* handles different variations of the same basic product. Hamer and Lepoeter claim these dimensions are quite simple when considered independently, but, in reality, many dimensions must be handled simultaneously resulting in many different nontrivial solutions. This framework is flawed, however, in that it leaves out the interaction between components in an aggregate model.

Zanella and Gubian [64] also describe a generalized model of a *design manager* that is "a set of functions which build, maintain, display, manage, and enforce relationships among the data and among the design tools which are involved in a project." The design manager controls the design software, supports the design methodology, coordinates large sets of data, maintains design integrity, and reacts to changes in the design environment. Zanella and Gubian break the functions required of a design manager into two groups. Static functions help in establishing and representing relationships among objects. Dynamic functions support design transformations that involve any kind of changes in the relationships among objects. Zanella and Gubian identify a number of design management relationships that are similar to those discussed by Katz (see Section 2.3) for aggregation, refinement, and equivalences. Zanella and Gubian emphasize the functions of a design manager, but provide only high-level conceptual requirements for the functions that should be performed.

While PDM systems are useful for controlling data and performing high-level product management tasks, they have many limitations. Design information such as functionality or geometric constraints can not be associated with the relationships between components. This means another tool must be used to document this information and additional steps taken to integrate the results back into the PDM model. Bilgiç and Rock point out that "PDM systems do not have a formal representation of the product that unambiguously describes its function. Most of the valuable information about the products stays in the "documents" the PDM system is managing."[4] This separation of the detailed information from the product structure adds complexity to the design model by requiring more links and it adds considerable overhead to the design process.

The document centered management approach of PDM systems restricts the designer's ability to implement and analyze incremental changes to small parts of a document. Instead, the designer must work with entire documents and manually ensure that related parts of other documents are kept consistent. Miller states that integration of PDM with other CAD applications continues to be a major challenge for PDM users [40]. As a consequence, some application models can not be completely incorporated into the meta-model, thus requiring additional steps to keep these separate representations consistent. Bilgiç and Rock identify other limitations of PDM systems including the inability to analyze the impact of proposed changes, the lack of capabilities to classify products by functionality, the inability to reuse design knowledge, and the inadequate support of resource, performance, and risk management [4].

Due to these limitations, PDM systems require considerable overhead to effectively integrate complex product designs and applications. This reduces the utility of PDM systems for incremental design changes, rapid design development, or small production runs. These limitations are reflected in many organizational implementations of PDM systems in which only the data vault and document management capabilities are utilized [23, 32].

## 2.5 Summary and Analysis

Considerable research has been performed to support the design and evolution of complex products. Feature-based design enables designers to manipulate standardized, reusable design abstractions rather than the individual points, curves, and surfaces of an entity's geometry. Features can also encapsulate additional design information such as functionality or manufacturing processes, although this capability has not been effectively exploited. Features can support concurrent design by representing design disciplines as different views. Mapping between views, however, is a difficult problem and current approaches require manipulation of the low-level geometry, thus negating the benefits of the higher-level feature abstraction.

A number of data models have been proposed for representing the hierarchical

aggregations and constraints of complex designs. Different models are proposed for functional or conceptual designs and for detailed, manufacturable designs. Because of the ambiguity associated with specifying functionality, the functional models are only able to capture a portion of the design functionality. The more detailed models have difficulty representing the detailed information associated with the interaction between parts in an assembly. Specialized relationships such as Lee's *mating features* facilitate specification of kinematic and positioning constraints [38], but provide no easy way to incorporate other functional or geometric information. Embedding complexity relationships in conceptual and detailed design models facilitates analysis and change propagation; however, it is difficult to develop a widely applicable modeling representation.

Version management systems have been implemented to track and control modifications to design components and some of these versioning systems support alternative designs, multiple views, and different configurations of a product design. Most version management systems, however, have only been implemented for the management of program text associated with software design. Since product structure is critical to manufacturing design, these systems are inadequate.

Product data management systems provide a comprehensive framework for managing all documents, applications, and processes that contribute to a product definition. PDM takes such a high-level approach, however, that the details of the design are not visible. This reduces the effectiveness of PDM in analyzing the design or implementing incremental changes. As a consequence, PDM systems are largely used only as secure document repositories.

# CHAPTER 3

# CASE STUDIES IN COMPLEX DESIGN

The design of complex products can be greatly facilitated by automating some design activities and assisting with other design tasks. Automation can enhance the management of product complexity by supporting the representation of product structure and functionality; by providing variable granularity, multiple views, and different abstractions of the product design; and by incorporating fasteners and connectors and design constraints into the product representation. Process activities that can benefit from automation include, among others, design decomposition, multidisciplinary analysis, simultaneous design, design reuse, controlled evolution, change management, and change propagation. These capabilities and activities are described in greater detail in Section 1.4.4.

This chapter presents case studies for the design of an automobile and a machining center that demonstrate the activities and capabilities identified above. Portions of these case studies appear as examples in this document to illustrate and explain the capabilities of the automated framework introduced in this research. The case studies are also used to analyze this research and compare the capabilities of other design tools and research (see Chapter 7).

## 3.1 Simultaneous Design of a Formula Automobile

Modern automobiles provide a comprehensive example of complex product design. A single automobile contains thousands of components allocated among many different subassemblies. Any particular car model may be available in many different configurations that are minor variations of the basic model. The industry is highly competitive so auto makers are pressured to develop innovative features

and new models in relatively short periods of time. There are many safety, environmental, and budgetary constraints that further limit the design options.

Each year, as part of a national engineering competition sponsored by the Society of Automotive Engineers (SAE), an undergraduate design class at the University of Utah designs, builds, and races a prototype formula automobile (FormulaSAE). While not as complex as a modern road vehicle, the design process and issues for this automobile provide a microcosmic demonstration of the challenges for computer-aided design.

This case study uses the models and processes developed for the formula automobile design class [14, 15, 57], but modifies them slightly to emphasize desirable complexity management capabilities that are identified in this research and supported by the resulting research tools. The general requirements and constraints of the formula automobile, as described in an overview of the project [57], are described below.

> For the purpose of the competition, the students are to assume that a manufacturing firm has engaged them to produce a prototype car for evaluation as a production item. The intended sales market is the nonprofessional weekend autocross racer. Therefore, the car must have very high performance in terms of its acceleration, braking, and handling qualities. The car must be low in cost, easy to maintain, and reliable. In addition, the car's marketability is enhanced by other factors such as aesthetics, comfort, and the use of common parts. The manufacturing firm is planning to produce 1000 cars per year at a cost under 8500 dollars. The challenge to the design team is to design and fabricate a prototype car that best meets these goals and [constraints].
>
> The primary restrictions on the design are the safety requirements and the engine size and intake restrictor. There is a minimum wheelbase of 1520 millimeters (60 inches) and the cars must have a working suspension with a minimum usable wheel travel of at least 50 millimeters (2 inches). The cars must also have four wheel brakes capable of locking all four wheels on dry asphalt at any speed. To ensure that the cars will not tip over during the performance events, the cars must not roll over when subjected to a tilt test with the car tipped to an angle of 57 degrees with the tallest driver in the car. Other safety requirements specify front and rear roll hoops, side impact protection, driver restraint systems, and driver safety equipment.
>
> The engine may be any four-cycle piston engine with a displacement of not more than 610 cubic centimeters. The fuels allowed are non-leaded premium gasoline, non-leaded 100 octane racing gasoline, and M85, a 85

> per cent methanol, 15 percent gasoline mixture. To limit the power of the engine, a single 20 millimeter diameter restrictor must be placed between the throttle and the engine for gasoline-fueled cars. For M85-fueled cars, the restrictor is limited to an 18 millimeter diameter. Supercharging or turbocharging is permitted provided that the restrictor is upstream from the supercharger or turbocharger. Any type of transmission or drive train may be used.

Due to the short time for development and the high complexity of the formula automobile, multiple design subteams work simultaneously on different portions of the overall design. With this approach, it is desirable that the design be decomposed into sections that are largely independent of each other.

To improve understanding and to organize the project for simultaneous design, the design team decomposes the formula automobile into smaller sections. The team initially identifies three major functional subsystems: the *body* that makes the automobile more aerodynamic and improves the appearance; the *chassis* that provides support, rigidity, and other functionality; and the *power train* that provides the power to move the car. The chassis and the power train are too complex to allocate to a single subteam, so the design team decomposes these two subsystems into smaller subassemblies as shown in Figure 3.1. The design team allocates each of the subassemblies identified in this figure to a subteam for further design and analysis. To avoid duplication and to simplify presentation, this case study uses only the rear portion of the automobile that includes the rear suspension, rear wheels, rear brakes, and the power train.

Each of the subsystems in Figure 3.1 has some sort of functional, geometric, or kinematic interaction with other subsystems in the automobile. For example, the rear suspension interacts with the brakes, the wheels, and the power train. To avoid difficulties when integrating the individually designed subsystems into the complete product design, the design subteams need to coordinate on these areas of interaction.

A considerable amount of design information is associated with the interactions between subsystems. As these interactions are agreed upon by various subteams, it is often helpful to document the resulting descriptions to minimize later misunder-

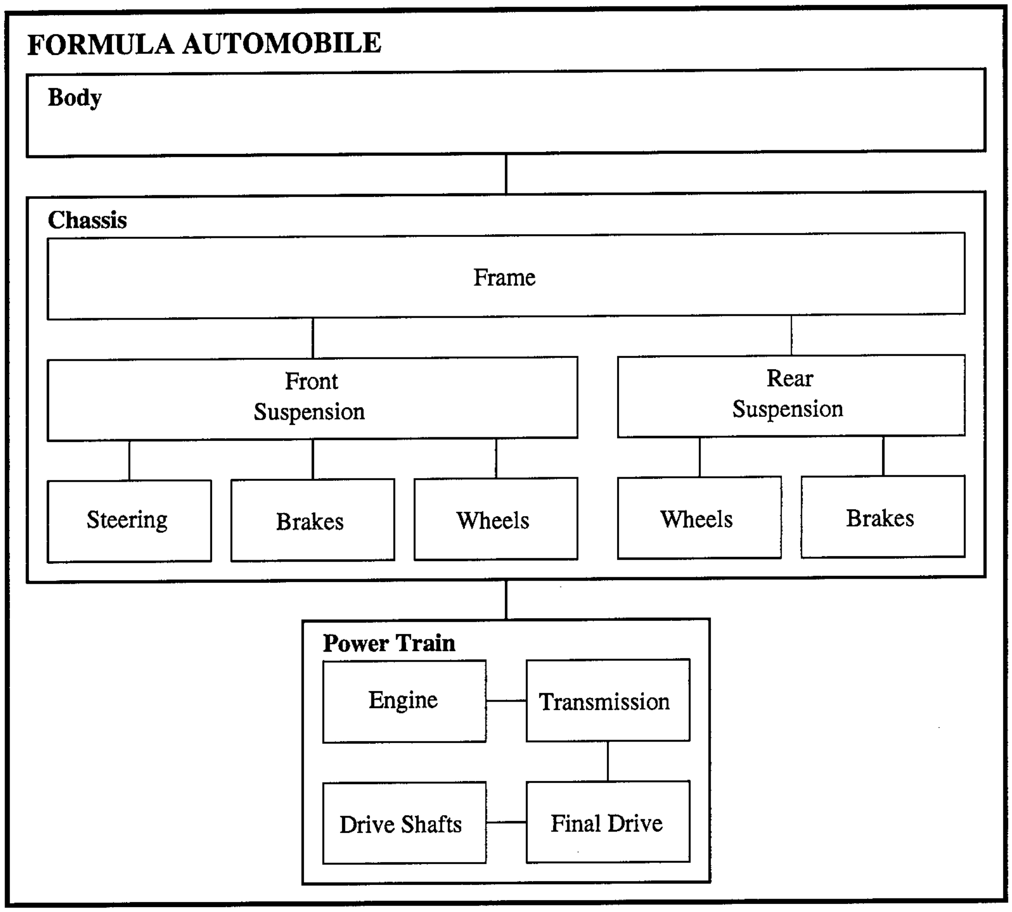



**Figure 3.1.** High-level decomposition of formula automobile

standings. If during the course of designing the independent components, a design subteam finds it impossible or too costly to conform to a previously agreed upon interaction description, the subteams must work together to modify the description.

The individual subteams proceed by decomposing the subsystems for the rear section of the automobile as shown in Figure 3.2. These subassemblies and parts are explained in the following paragraphs.

The wheel transfers power from the power train to the road to move the car forward. Wheel sizes are standardized to accommodate tires. The wheel is connected to the hub with a standardized arrangement of bolts.

The rear suspension supports the weight of the car and provides stability to the ride. The rear suspension contains support members, springs, and shock absorbers that provide torsional stability and allow limited vertical motion to absorb road

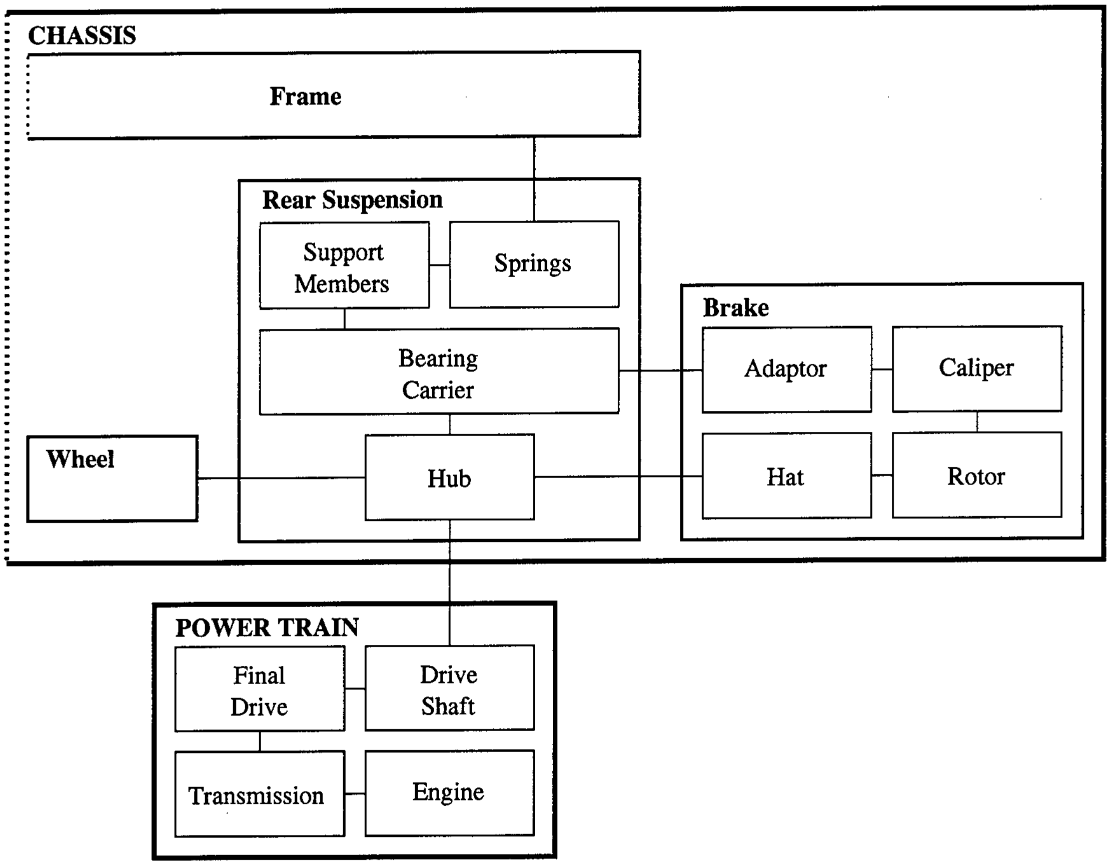



**Figure 3.2.** Decomposition of rear section of chassis and power train

bumps; a hub to which the wheel and the drive shaft are connected; and a bearing carrier to support the hub while allowing the hub to rotate. As shown in Figure 3.2, the decomposition of the rear suspension also creates additional areas of interaction, both within the rear suspension subassembly and between the components of the rear suspension and other subassemblies, that the designers must coordinate to ensure compatibility with other parts. The common areas of interaction with other subassemblies now include the transfer of power, through the hub, from the drive shaft to the wheel; the connection of the springs to the frame; and the support of the brake adaptor with a bolted connection to the bearing carrier.

The brake subassembly brings the entire car to a stop by halting the rotational motion of the wheels. A significant concern of the braking subsystem is dealing with the considerable heat that is generated from the frictional forces. For the disk

brakes used in the formula automobile, the wheel is stopped with a caliper that squeezes a rotor until the friction stops the rotor from rotating. In addition to the rotor and the caliper, the brake subassembly contains a brake hat to which the rotor is connected and an adaptor that supports the caliper. The brake subassembly interacts with the rear suspension through a bolted connection between the brake hat and the hub, and through a bolted connection between the adaptor and the bearing carrier.

The power train generates power and transforms it into torque that is applied to the drive shafts. The power train contains an engine, transmission, and drive shafts that are purchased from other manufacturers. Their dimensions, requirements, and performance specifications are integrated into the automobile design. The power train also includes the final drive that transforms the power from the engine and transmission into the desired torque to apply to the drive shafts. The drive shaft is supported through its interaction with the rear suspension hub.

The design of each subsystem requires expertise in a number of different areas to include design functionality and ease of manufacture and assembly. By concurrently analyzing and designing for these different areas, design subteams can improve their efficiency.

As shown in Figures 3.1 and 3.2, different subsystems can be decomposed at different levels of detail. Also, since each subteam proceeds at a different rate, the subteams must be able to simultaneously work at different levels of detail [12].

The University of Utah enters the design competition each year with a completely different set of students. Students are better able to reuse or adapt components from previous versions of the automobile if they are able to understand the rationale and history that led to the previous designs and if they can easily recover these old designs and modify them as necessary.

## 3.2 Incremental Design of a Machining Center

Because of the amount of rework that results from design changes made late in the design process, errors caught early in the design process are generally easier

and less costly to correct. An incremental design process, if properly implemented and supported, would increase the chance of early error detection by performing multidisciplinary analysis over small increments rather than after complete design phases as in traditional waterfall models. Small increments reduce the complexity of design analysis, yet they allow the designer to consider the entire model during this analysis to ensure compatibility and completeness. Trying different alternatives is less costly since the designer can control the level of detail in each alternative. Figure 3.3 presents a pseudo-algorithm for the incremental design process used in this case study.

A machining center creates a manufactured part by cutting away excess material from a standardized piece of stock. Machining centers are highly complex with thousands of complex parts and precise operating constraints. Speed, accuracy, and cutting tool access are critical requirements of the milling operations. Although many machining centers already exist, this case study explores alternatives of a

1. SPECIFY EXTERNAL INTERACTIONS AND CONSTRAINTS
2. DECOMPOSE INTO SUBASSEMBLIES OR COMPONENTS
3. SPECIFY INTERNAL INTERACTIONS AND CONSTRAINTS
4. DESIGN SUBASSEMBLIES OR COMPONENTS
5. ANALYZE
6. IF SATISFIED, THEN QUIT
7. OTHERWISE, REFINE IN ONE OF THE FOLLOWING WAYS:
   - 7.1. MODIFY AT SAME ABSTRACTION LEVEL
     - 7.1.1. MODIFY EXISTING SUBASSEMBLIES, COMPONENTS, OR INTERACTION CONSTRAINTS
     - 7.1.2. GO TO 5
   - 7.2. ADD AT SAME ABSTRACTION LEVEL
     - 7.2.1. ADD ADDITIONAL SUBASSEMBLIES OR COMPONENTS
     - 7.2.2. GO TO 3
   - 7.3. DECOMPOSE AT LOWER LEVEL OF ABSTRACTION
     - 7.3.1. MAP INTERNAL INTERACTION CONSTRAINTS TO EXTERNAL INTERACTION CONSTRAINTS, AS NECESSARY
     - 7.3.2. GO TO 2

**Figure 3.3.** Incremental design process

particularly complex subassembly of the machining center, the spindle cartridge, for innovations that could increase performance. The design is real; however, the innovative design scenario is simulated to emphasize exploratory aspects of the design process that are well suited to incremental design.

The design team initially divides the machining center into six subassemblies as shown in Figure 3.4: a spindle head for mounting and spinning the cutting tools, a drive for moving the spindle head in a vertical direction, a column for mounting the vertical drive, a table for mounting the work piece, an X-Y drive for moving the workpiece horizontally, and a bed upon which the column and the X-Y drive are mounted [56]. As shown in Figure 3.5, the spindle head is further decomposed into a spindle cartridge that holds and rotates the cutting tool, a spindle drive that provides the power to spin the spindle cartridge, and a head casting upon which the cartridge is mounted. In this case study, the incremental design process in Figure 3.3 is used to explore innovative designs for the spindle cartridge .

As the first step in the incremental design process, the designer identifies the operating environment for the spindle cartridge and specifies the external constraints imposed on the cartridge. These constraints include the size of the tools, the required milling accuracy, and the desired cutting speed. Although not part of the spindle cartridge, the tool holder is a standardized part that further constrains the design of the spindle cartridge. The tool holder holds cutting tools that interact with the part being milled, thereby exerting forces on the spindle cartridge subassembly. These forces, along with the interaction of the spindle cartridge with the head casting and the spindle drive, must all be considered in the design of the spindle cartridge. To accommodate these external constraints, the design team specifies the interaction of the spindle cartridge with the tool holder, the head casting, and the spindle drive.

After the external constraints are specified, the design team determines the major functions or concepts in the cartridge design and converts these concepts into the initial design components. The spindle cartridge is decomposed into three major functional components: a spindle that rotates at a high rate of speed, a

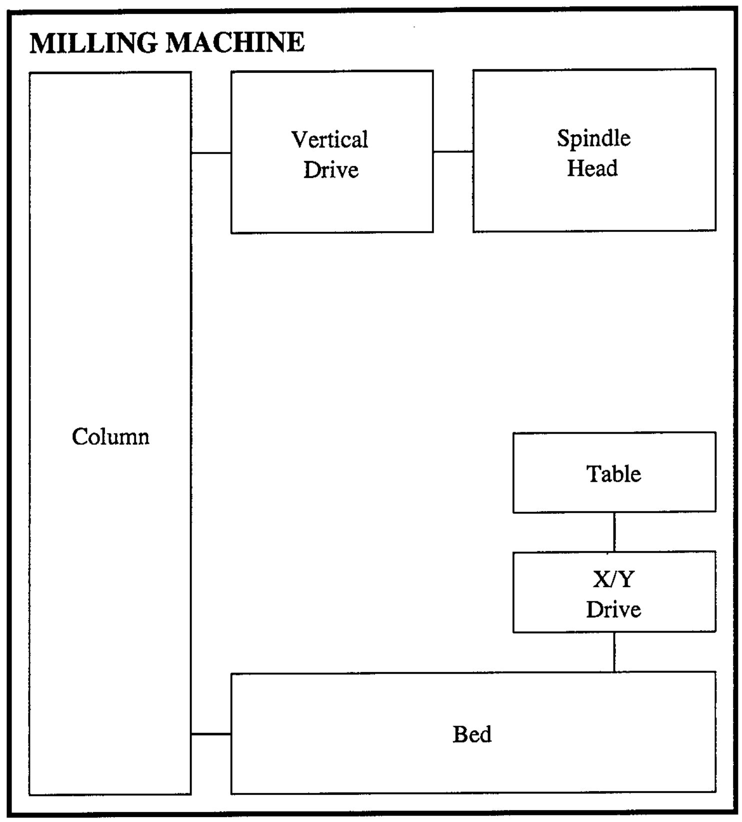



**Figure 3.4.** Initial decomposition of machining center

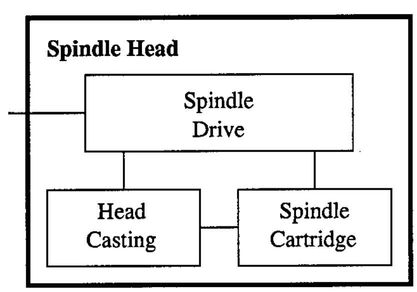



**Figure 3.5.** Additional decomposition for spindle head

housing to provide a stable mounting for the spindle, and a draw bar for mounting the tool holder. As the design team decomposes the spindle cartridge they also identify interactions between the spindle, the housing, and the draw bar.

The design team continues the incremental design process with the high-level design of the major components within the spindle cartridge subassembly. Because of the innovative nature of the design, the design team needs to create different variants of the design and analyze each variant with respect to functionality, manufacturability, and ease of assembly. Each variant is subject to the same interaction constraints as the original design. When the initial high-level design increment is completed, the designers need to analyze the design for conformance with constraints and to determine how to proceed with the next increment.

The design team has a number of options for refining the design. If analysis reveals discrepancies in the design, the designers could correct these by modifying parameters or constraints. Alternatively, the design team could add additional components, at the same level of detail, to satisfy missing functionality. Once a satisfactory design is obtained at one level of detail, the design team could further decompose the design by adding additional detail and constraints.

At any detail level, different design teams may need to simultaneously design independent design components or subassemblies. Design teams should modify and reuse existing design components where possible.

# CHAPTER 4

# AGGREGATION

Given a human's relatively fixed capacity for designing, understanding, and interpreting large, complex scenarios, designers have developed methods of organizing the vast amounts of information inherent to a complex product design. This organization usually results in a hierarchical structure with relations identifying entities that are part of a higher-level composite entity [31]. In common English usage, an *aggregation* is defined as "a group, body, or mass composed of many distinct parts; an assemblage," and the term *aggregate* is defined as "a mass or body of units or parts somewhat loosely associated with one another." For design usage, this research defines an *aggregation* to be a composite design entity along with its relations with other entities. Aggregations are necessary to organize complex design data into a comprehensible product structure. Design systems that strive to help designers manage complexity, not just represent it, then, must provide mechanisms to help designers create, store, query, and modify such structures and relations.

Because of its importance in organizing complex design models, aggregation is supported to varying degrees in many existing CAD tools. In a typical implementation, an aggregation is simply a notational convenience for accessing multiple existing objects. Recent research, however, has extended the aggregation concept to more comprehensive product models. Gui and Mäntylä's *multigraph* [27] provides a conceptual model of the design hierarchy that emphasizes the functionality of the design components. Product data management (PDM) systems [4, 40, 42] focus on the structure of the design model with hierarchical links between major design documents. In the multigraph and PDM representations, detailed manufacturing information is modeled separately and then linked into the hierarchical aggregation

structure. Other aggregation models [17, 38] embed detailed manufacturing information directly into the hierarchical model structure, but restrict the designer's ability to move among the levels to independently manipulate the different design abstractions.

Recognizing the dynamic nature of design, this research views aggregation as a tool for representing and managing the evolution of a design. Aggregations are used both to capture the organizational structure of a design model and to integrate diverse, multidisciplinary design information into a single entity. Rather than limiting evolution to a top-down decomposition, this research uses aggregation to increase the flexibility for organizing design data into multiple levels of abstraction and at different levels of granularity in a form that is useful to a design team's process. Individual design components can be independently analyzed and refined while still belonging to a higher-level aggregation.

This research presents several aggregation and relationship concepts, along with their implemented objects, that together make it easier for a designer to flexibly organize, analyze, and evolve a complex design.

- The *attachment* describes the hierarchical relationships between design components. That is, "A is part of B".

- The *interface specification object* specifies peer-to-peer relationships between components. It has all constraints, interdependencies, and other information that a component needs to interact with another component.

- The *neighborhood*, *part*, and *assembly* aggregations each organizes multiple components into a single design object.

*Attachments* and *interface specifications* each create a logical relationship between two existing design components. Each is implemented as an independent design object having references to the related components. Additional information relevant to the relationship, such as relative positioning constraints between the two components, is incorporated into the relationships objects, rather than into

the individual components. This isolates the related components from each other so that each component may be designed independent of the other.

The objects implemented to create *neighborhood*, *part*, and *assembly* aggregations each creates a scope that contains all lower-level components in the aggregation object. Access to the lower-level components is possible only by entering the aggregation scope in which they are defined. As their names suggest, these elements represent three types of clustering. The *neighborhood* aggregation organizes components into a single entity, but requires no further relationships between the lower-level components. The *part* aggregation requires that components be linked with *attachments* indicating the components that are "part of" the aggregation. Similarly, the *assembly* aggregation requires that components be linked with *interface specification objects* that specify how the parts of the assembly interact.

## 4.1 Role in Complexity Management

A complex product design is comprised of many different components representing different views of the design at multiple levels of detail. To form a working product design, low-level features and geometry objects are linked together to form manufacturable parts. These parts work together to form a functional assembly. Multiple assemblies may work together in a complete product. Aggregation is the mechanism for structuring these components into recognizable parts or assemblies.

From a top-down perspective, designers initially lay out the design at a high level with a limited number of components. The high-level design could be a conceptual design with functional descriptions and no geometry or it could be an initial design layout with rough geometry such as cylinders and boxes. By analyzing simple conceptual models, designers can narrow down the set of possible design solutions without expending considerable resources. As additional detail is defined, designers decompose components from the high-level layout into multiple subcomponents.

This research also uses aggregation as a mechanism for controlling changes to a design model. By implementing aggregations as independent design modules that encapsulate low-level design information, other design entities have a limited view of

the design data within the aggregation. The aggregation forms a boundary to give the designer control over how external changes affect the design entities within the aggregation and how internal changes to the aggregation are propagated to external design entities. This modularity facilitates change management by providing a well defined entity to control and an entity with which to associate design rationale and history. The designer can restrict changes to remain within the boundaries of the aggregation, thereby facilitating simultaneous development of different aggregations by different design teams and enabling reuse of these aggregations in other product designs.

A representation for assembly aggregations, in particular, is important for extending modeling capabilities beyond the specification of low-level geometry and manufacturing features. The assembly aggregation provides the necessary framework for laying out a design model at conceptual, functional, and detailed levels of abstraction. Because product functionality is realized by the interaction of different components rather than by individual components alone, Sodhi and Turner suggest that an assembly-modeling framework is the key for a design environment that can capture and maintain functional intent [51]. Shah and Mäntylä describe additional benefits of assembly modeling to include interference detection between parts, motion simulation, constraint satisfaction, assemblability evaluation, and assembly manufacturing planning [49].

## 4.2 Underlying Concepts and Terminology

Some standard terminology is used repeatedly throughout this document to refer to particular implementation concepts or structures. Although minor in their support of the thesis explored in this research, these concepts are necessary for understanding some of the larger, more significant concepts and structures.

### 4.2.1 Design Objects and Constructors

A *design object* refers to any design modeling entity that can be created and manipulated within the modeling environment. For example, design objects include, among others, curves, surfaces, and other geometric entities, features, mathematical

models or constraint entities, composite structures, or textual and numeric entities. The term "design object," however, does not imply a classification or inheritance structure as it does in object-oriented software design, although, if supported by the modeling environment, objects with inheritance and classification would also fit the definition for "design objects."

The modeling command that creates a design object is called a *constructor*. The examples in this research use the text-based design specification language of *Alpha_1*, however, menu and pointer commands from a graphical user interface are equally applicable as constructors if they result in the creation of a design object.

### 4.2.2 Positioning Constraints

Positioning of related design objects within a complex product design model is essential for visualizing and analyzing the design. In this research, positioning constraints are specified with user defined anchors that describe a local coordinate system in *Alpha_1*. Each object to be positioned must have a positioning anchor associated with it. Many design objects in *Alpha_1* include positioning anchors in their definition. This research also allows other design objects to be associated with a positioning anchor by wrapping these objects within a design object that provides an anchor. To position an object, the designer specifies the desired position with another positioning anchor, and the object is aligned with this anchor using existing routines from *Alpha_1*.

This positioning mechanism is used to automatically align features, parts, and subassemblies when part or assembly aggregations are created. A change in one positioning constraint will be automatically propagated throughout the entire aggregation. For example, changing the positioning constraint for a hole feature will cause the hole to be relocated along with any component linked to the hole. If a shaft has been inserted into the hole, the shaft will be relocated, as will any other components linked to the shaft.

## 4.3 Aggregation Relationships

Although design data is hierarchical, there are two types of relationships necessary to define an aggregation – hierarchical parent-child relationships and a peer-to-peer relationships that define how components at the same level interact with each other. These relationships are implemented in this research as *attachments* and *interface specification objects*, respectively.

The hierarchical relationships in an aggregation are represented with *attachment* relations that link a child component to its parent (Figure 4.1). Since the primary representation of mechanical design data is geometric, the *attachment* relation includes location information for positioning the child component relative to its parent.

Peer-to-peer relationships are defined at the assembly level between interacting parts or subassemblies. Designers specify peer-to-peer relationships in *interface specification objects* that link two parts and provide positioning and kinematic constraints along with other relevant design information as described in Chapter 5.

In many design tools, information associated with the relationships between components is embedded into the linked components. In this research, however, the *attachment* and *interface specification object* are independent design objects that the designer can manipulate independent of the entities that are linked by these relationships. The designer can use the attachment and interface specification

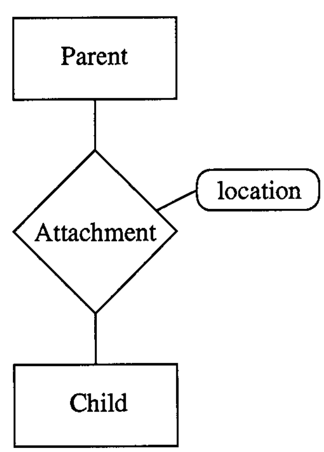



**Figure 4.1**. The parent-child relation depicted by an *attachment*

relationships to manipulate the model structure or configuration without modifying the actual design entities. With this independence, designers can reuse preexisting component designs and integrate them into a new design model without having to modify the original component.

## 4.4 Aggregation Objects

An aggregation object is a single entity, representing a portion of the hierarchical model graph, that encapsulates detailed information into a single module while separating this information from other parts of the model. This research defines three types of aggregation objects for grouping mechanical design information – *neighborhoods, parts*, and *assemblies.*

Only a limited amount of information is required to be specified for each type of aggregation object. Thus, designers can create an aggregation initially with only a small amount of conceptual information. Designers can then add further detail to the aggregation as the design evolves. With this approach, designers have considerable flexibility on how they evolve a design, yet the aggregation objects ensure that the designers provide sufficient information to link related design components.

Designers may modify and reconfigure an aggregation to contain different components for different levels of abstraction or for different views. For example, the functional abstraction of the formula automobile configures brake parts into a single aggregation, while a separate abstraction representing rigid subassemblies divides the brake parts into two separate aggregations. Designers can include an object in more than one aggregation by separately defining the object and inserting a copy into the appropriate aggregations.

### 4.4.1 Neighborhood

A *neighborhood* is a generic aggregation for isolating and modularizing any group of objects in the design model. No explicit relationship is required among members of a neighborhood – the designer simply needs to identify those design objects that should be in a particular neighborhood.

A neighborhood is useful for encapsulating low-level geometric entities or parameters or as a temporary place holder during early stages of the design when the relationships between design entities are not yet well specified. A neighborhood is the least structured of the three aggregations and is generally applicable in all situations where the more structured part or assembly aggregations are inappropriate or where insufficient information is available to describe the relationships required in the part and assembly aggregations.

A neighborhood may be nested within another neighborhood to form a hierarchical structure with increasingly more detail at the lower levels. When used in this fashion, however, there are no explicit links, such as attachments, to describe the relationships between the two levels.

In the formula automobile, neighborhoods are used by the designers to represent the low-level parameters, points, lines, and circles, from which complex curves and surfaces are derived. Figure 4.2 shows the *Alpha_1* design specification language description of a neighborhood that encapsulates the geometry of the heat slots in the automobile's brake rotor. In this example, the neighborhood is identified with the *seq* constructor that incorporates all the design objects within the brackets into a single neighborhood.

Once an object is identified as a member of a neighborhood, access to that design object, from outside of the neighborhood, is restricted. Reference to the neighborhood object itself provides access to the last object identified in the neighborhood or a collection of all objects in the neighborhood, as selected by the designer. In Figure 4.2, the actual heat slot surface, *Slots*, is the last object in the sequence and is directly accessible by referencing the neighborhood variable, *HeatSlots*. The complex surface for the heat slots is shown in Figure 4.3. By assigning neighborhood objects with the "`:*`" operator, designers can also make objects within the neighborhood accessible on an individual basis by indirect reference through the neighborhood object. For example, the object associated with *SlotCir1* is accessible from outside of the *HeatSlot* neighborhood through the constructor *HeatSlots::SlotCir1*. This constructor creates a copy of the *SlotCir1*

```
HeatSlots : seq{

    "Slots";

    CirDia : ( Rotor_Hat_Intfc::BoltCirDia +
                  RotorParms::SlotCirOffset );
    CtrPt : pt( 0.0, -RotorParms::SlotCtrRad );
    CtrCir : circleCtrRad( CtrPt, RotorParms::SlotCtrRad );
    ConstLine1 : linePtAngle( CtrPt, 35.0 );
    Point1 : ptIntersectCircleLine( CtrCir, ConstLine1, false );
    Cir1 :* circleCtrRad( origin, RotorParms::SlotWidth/2.0 );
    Cir2 :* circleCtrRad( Point1, RotorParms::SlotWidth/2.0 );
    Cir3 :* circleRadTan2Circles( RotorParms::SlotCtrRad +
                                   RotorParms::SlotWidth/2.0,
                                   cir1, cir2,
                                   true, true, true );
    Cir4 :* circleRadTan2Circles( RotorParms::SlotCtrRad -
                                   RotorParms::SlotWidth/2.0,
                                   cir1, cir2,
                                   false, false, true );
    SlotCrv : outlineCrv( array( Cir1, Cir3, Cir2,
                                 CircCCw( Cir4 )),
                          false );
    Slot : profileSide( SlotCrv, "inside",
                        RotorParms::Thick + 0.1, 0.0, 0.0, 0.0 );
    Anchor1 : rotateAnchor( Prims::Anchor1, 0.0, 0.0,
                            -RotorParms::SlotFrstAng );
    SlotPattern1 : RadialPattern( Anchor1, Slot,
                                  RotorParms::SlotNum,
                                  (RotorParms::SlotFrstAng+90.0)/
                                    RotorParms::SlotNum, CirDia );
    SlotPatternAnchor2 : rotateAnchor( Anchor1,
                                        0.0, 0.0, 90.0 );
    SlotPattern2 : RadialPattern( Anchor2,
                                   Slot,
                                   RotorParms::SlotNum,
                                   (RotorParms::SlotFrstAng+90.0)/
                                   RotorParms::SlotNum,
                                   CirDia );
    Slots : entity( mergeShell( SlotPattern1,
                                SlotPattern2 ) );
    };
```

**Figure 4.2.** Specification of the heat slot surface

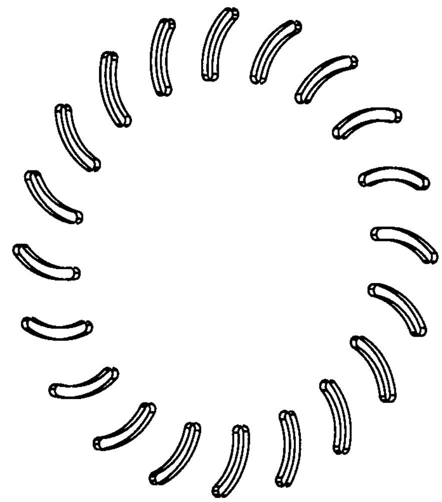


**Figure 4.3.** Heat slot surface

object so that any modifications will not affect the design object embedded in the neighborhood. Objects within the neighborhood may not be modified from outside of the neighborhood.

The neighborhood aggregation structure for the heat slot surfaces is shown in Figure 4.4. Although this structure shows a hierarchical relation among the design components, these relationships simply illustrate the dependencies of the object constructors. In an *Alpha_1* design model, any design object that uses another design object in its construction establishes a precedence relationship in which the

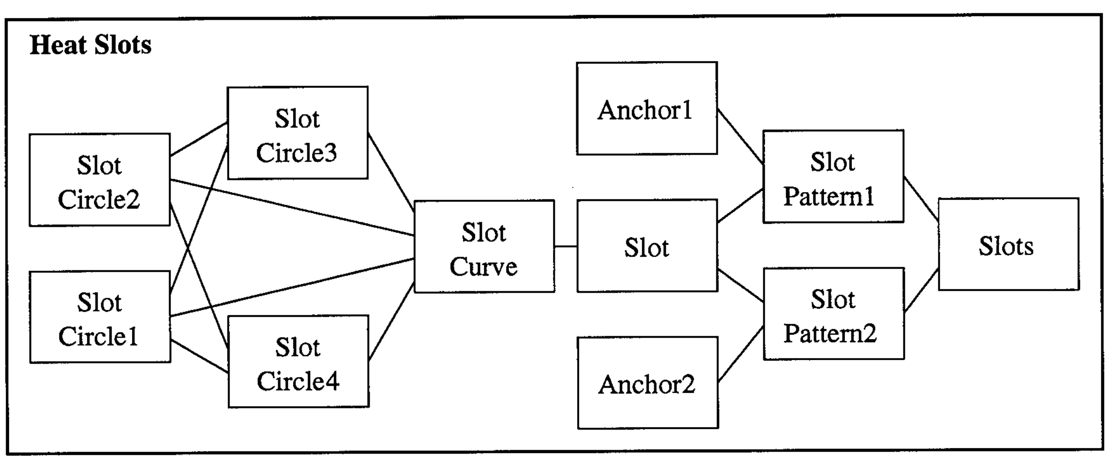



**Figure 4.4.** Structure of the heat slot neighborhood

new object becomes dependent on the previous object. No other information is associated with these relationships. The neighborhood structure could just as well be a group of objects with no dependencies between them. This is often the case when a neighborhood is used to encapsulate a set of numerical parameters.

### 4.4.2 Part

A *part* aggregation consolidates related design components that represent a single part. Part aggregations examine the *attachment* relationships between design objects to determine the hierarchical structure of the part and to position and validate the components of the part in accordance with specified attachment relationships.

An intent of the aggregation structure is to enable the designer to start out with a conceptual abstraction of a part and evolve this concept into a detailed part design by attaching additional features and geometry objects. Thus, in order to specify a part aggregation, one design object with a positioning anchor is designated as the base component. Then, additional components can be incorporated into the part aggregation with attachment relationships.

The part aggregation is used in the formula automobile design model to represent all distinct parts. The brake rotor specified in Figure 4.5 is an example of one of these parts. Designers specify the part aggregation with the *partSeq* constructor that extends the neighborhood sequence to incorporate all design objects linked with *partof* attachments into a *part object.* The part object positions all attached components with respect to a selected base component and organizes the aligned components into a hierarchical structure. The part object is the accessible result of the *partSeq* constructor. The geometric representation of the brake rotor part object is shown in Figure 4.6.

The brake rotor part in Figure 4.5 consists of manufacturing features that indicate sections of the machining stock that must be cut away to form the final part. The *Stock* entity is identified as the base object by its position as the first object designated as a parent in a *partof* attachment. The designer organizes the part model by nesting the manufacturing details into neighborhood aggregations for each of the major features, *IDCut, OuterCut, CutOutPattern,* and *BoltHoles*,

```
BrakeRotor : partSeq{

    "Detailed design of the Rear Brake Rotor Model";
    Prims seq{
        . . .
    }
    Stock seq{
        . . .
    }
    IDCut seq{
        . . .
    }
    atch1 : partof( Stock, Prims::Anchor1, IDCut );

    OuterCut seq{
        . . .
    }
    atch2 : partof( Stock, Prims::Anchor1, OuterCut );

    HeatSlots : partSeq{

        "Heat Slots";
        SlotPattern1;
        atch1 : partof( OuterCut, Prims::Anchor1, SlotPattern1 );
        SlotPattern2;
        atch2 : partof( OuterCut, Prims::Anchor2, SlotPattern2 );
        SlotPattern3;
        atch3 : partof( OuterCut, Prims::Anchor3, SlotPattern3 );
        SlotPattern4;
        atch4 : partof( OuterCut, Prims::Anchor4, SlotPattern4 );
    }

    CutOutPattern seq{
        . . .
    }
    atch4 : partof( Stock, Prims::Anchor1, CutOutPattern );

    BoltHoles : Rotor_Hat_Intfc.pos_entity;
    atch5 : partof( stock, Prims::Anchor5, BoltHoles );
};
```

**Figure 4.5**. Specification of the brake rotor part

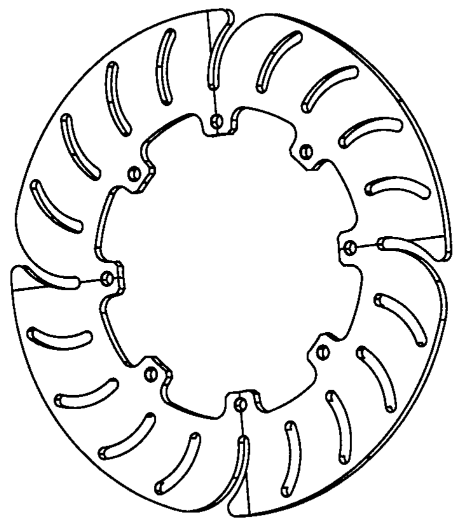


**Figure 4.6.** Brake rotor part

and attaching these features directly to the *Stock*. Since heat dissipation is a key functional characteristic of the brake rotor, the designer specifies the *HeatSlots* as a nested part aggregation that is further decomposed into four patterns that are attached to the *HeatSlot* to form an intermediate level of detail. This abstraction facilitates exploration of the heat dissipation capabilities of the brake rotor by allowing the designer to independently manipulate and analyze the heat slots. The structure of the brake rotor part is depicted in Figure 4.7. All objects that are attached to the *Stock*, either directly or through intermediate objects, are aligned and incorporated into the brake rotor part object.

To specify constraints or design goals, designers can embed additional design objects into the part aggregation without direct links into the attachment hierarchy. In the brake rotor part, the designer nests dimensional parameters, low-level geometric entities such as lines and circles, and other constraints in the *Prims* neighborhood aggregation. None of these objects is linked into the brake rotor part with an attachment relationship; however, by inclusion in the part specification, these entities become part of the part aggregation. The designers mark each key entity within the *Prims* aggregation for access by other components in the brake rotor part aggregation.

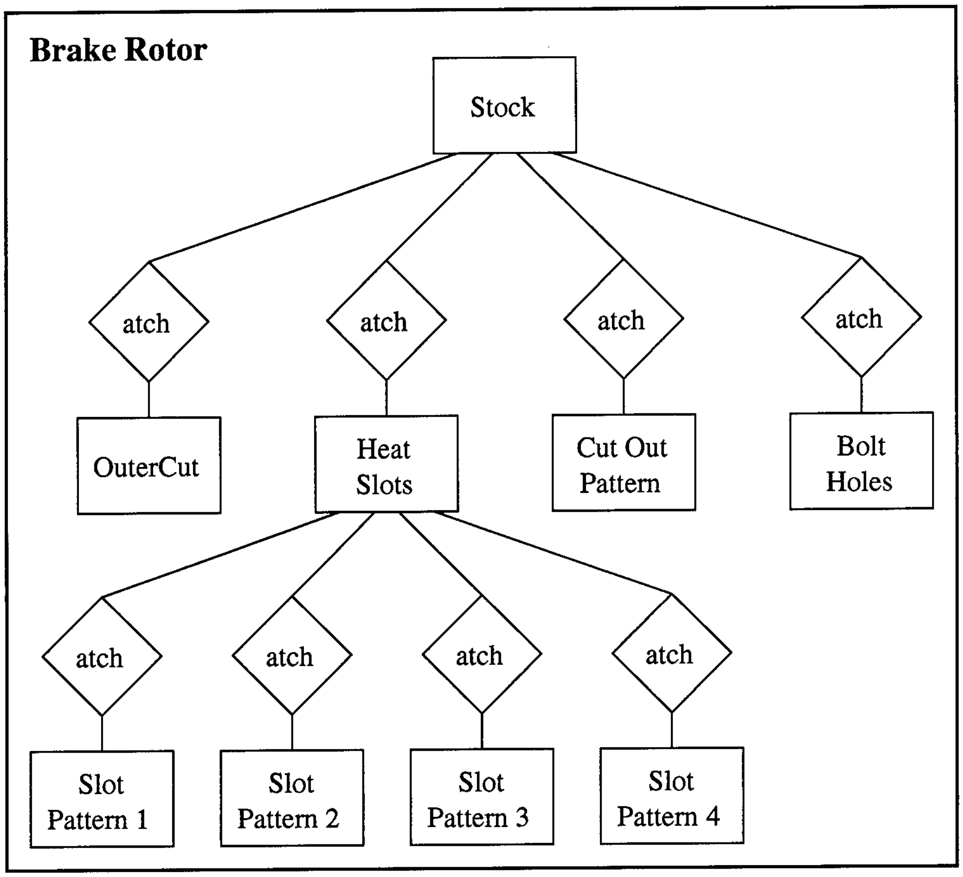



**Figure 4.7.** Structure of the brake rotor part aggregation

### 4.4.3 Assembly

An *assembly* aggregation organizes interacting parts or subassemblies into a single mechanical assembly. Assembly aggregation objects examine the *interface specification* relationships between parts to align the parts and to validate the parts in accordance with the interface constraints. In this research, the structure and methods provided by the assembly aggregation support high-level design layout, grouping and positioning of low-level parts, and multiple assembly configurations so that different types of analysis can be performed.

A minimal assembly aggregation contains two parts or subassemblies and an interface specification between them. To incorporate additional parts or subassemblies into the assembly aggregation, designers must create interface specification objects to link new components with another part in the assembly.

The formula automobile assembly is comprised of a number of subassemblies and parts as shown in Figure 4.8. As shown in this example, an assembly aggregation is specified with the constructor *assemblySeq* that extends the neighborhood sequence to incorporate design objects linked with interface specifications into an *assembly object.* An assembly object positions all interacting parts with respect to a selected base part and organizes the aligned parts into a list structure. The assembly object is the accessible result of the *assemblySeq* constructor.

Designers can use software assistants with the assembly object to perform automated analysis of the kinematics and forces associated with the interfaces in the assembly. For example, to analyze the forces acting on the formula automobile assembly, the designer invokes the command:

```
validateForces( FormulaSAE )
```

which examines the forces attached to the assembly aggregation and validates these forces against the specified interface constraints. Positioning of assembly components is automatically validated by the constructor as the assembly is updated. The designer can manipulate component positions to perform kinematic analysis of the assembly.

By not restricting the types of objects that can be linked into an assembly, designers can use generic entities with little design information at the conceptual level and detailed subassemblies or parts at more detailed levels. In the formula automobile example shown in Figure 4.8, the *Hat*, *Rotor*, *Wheel*, or other parts in any of the subassemblies can be specified independent of the assembly aggregation. These objects may represent a high-level conceptual decomposition as shown in Figure 4.9 or detailed information and geometry as depicted in Figure 4.10. If no existing definition of an object exists, the designer can choose to have a high-level placeholder object automatically created.

Like a neighborhood aggregation, the designer can embed additional design objects, such as parameters or textual descriptions, into the assembly aggregation without directly linking these objects to the assembly structure. These design

```
RearLayout : assemblySeq{

    "Layout of rear section of formula automobile";

    Chassis : assemblySeq{

        "Rear section of chassis";

        Frame;
        Wheel;
        RearSuspension : assemblySeq{
            BearingCarrier;
            Hub;
            Carrier_Hub_Intfc;
        }
        Brake : assemblySeq{
            Hat;
            Rotor;
            Caliper;
            Adaptor;
            Hat_Rotor_Intfc;
            Rotor_Caliper_Intfc;
            Caliper_Adaptor_Intfc;
        }

        Frame_RearSuspension_Intfc;
        RearSuspension_Wheel_Intfc;
        Brake_RearSuspension_Intfc;
    }

    PowerTrain : assemblySeq{
        DriveShaft;
        FinalDrive;
        Transmission;
        Engine;
        DriveShaft_FinalDrive_Intfc;
        FinalDrive_Transmission_Intfc;
        Transmission_Engine_Intfc;
    }

    Chassis_PowerTrain_Intfc;
};
```

**Figure 4.8**. Assembly specification for rear layout of formula automobile

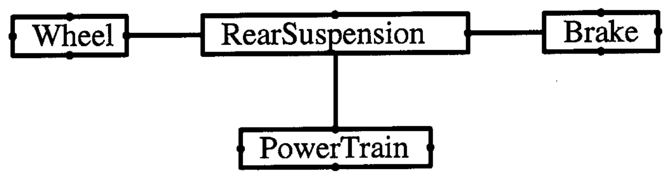



**Figure 4.9.** Conceptual decomposition of rear layout

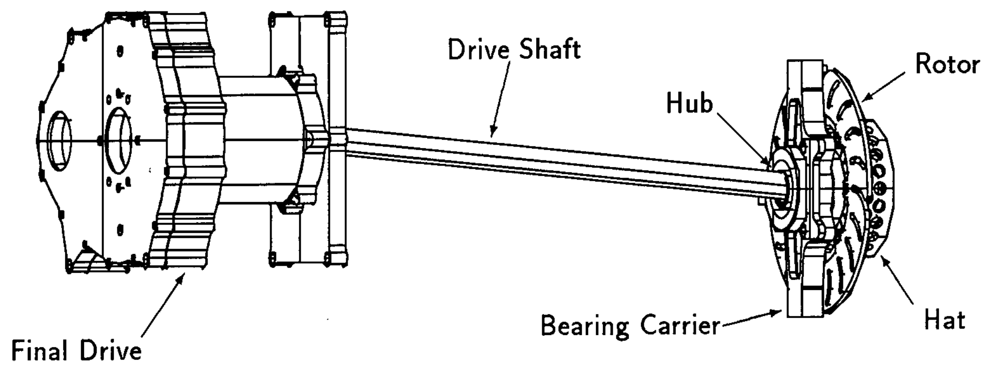



**Figure 4.10.** Detailed representation of rear layout with *Wheel* omitted

objects allow the designer to specify high-level constraints for the assembly components or to describe the purpose of the assembly as depicted in Figure 4.8.

The rear layout specified in Figure 4.8 has two major subassemblies – the power train and the chassis. To represent different levels of detail or additional subassemblies, the designer can nest assembly aggregations, such as those for the frame, rear suspension, brake, and the wheel within another assembly aggregation. The designer links each part or subassembly within the rear layout assembly to other parts or subassemblies using interface specification objects. The assembly structure represented by the rear layout specification is shown in Figure 4.11. Observe that the entities and relationships in this diagram directly parallel the independent conceptual decomposition of the rear layout depicted in Figure 3.2.

An *assembly* aggregation consolidates interacting parts or subassemblies into a complete product assembly model. Using the assembly aggregation, subassemblies can be further decomposed into additional subassemblies or parts. Designers can then create *part* aggregations that organize related features into a single part. Low-

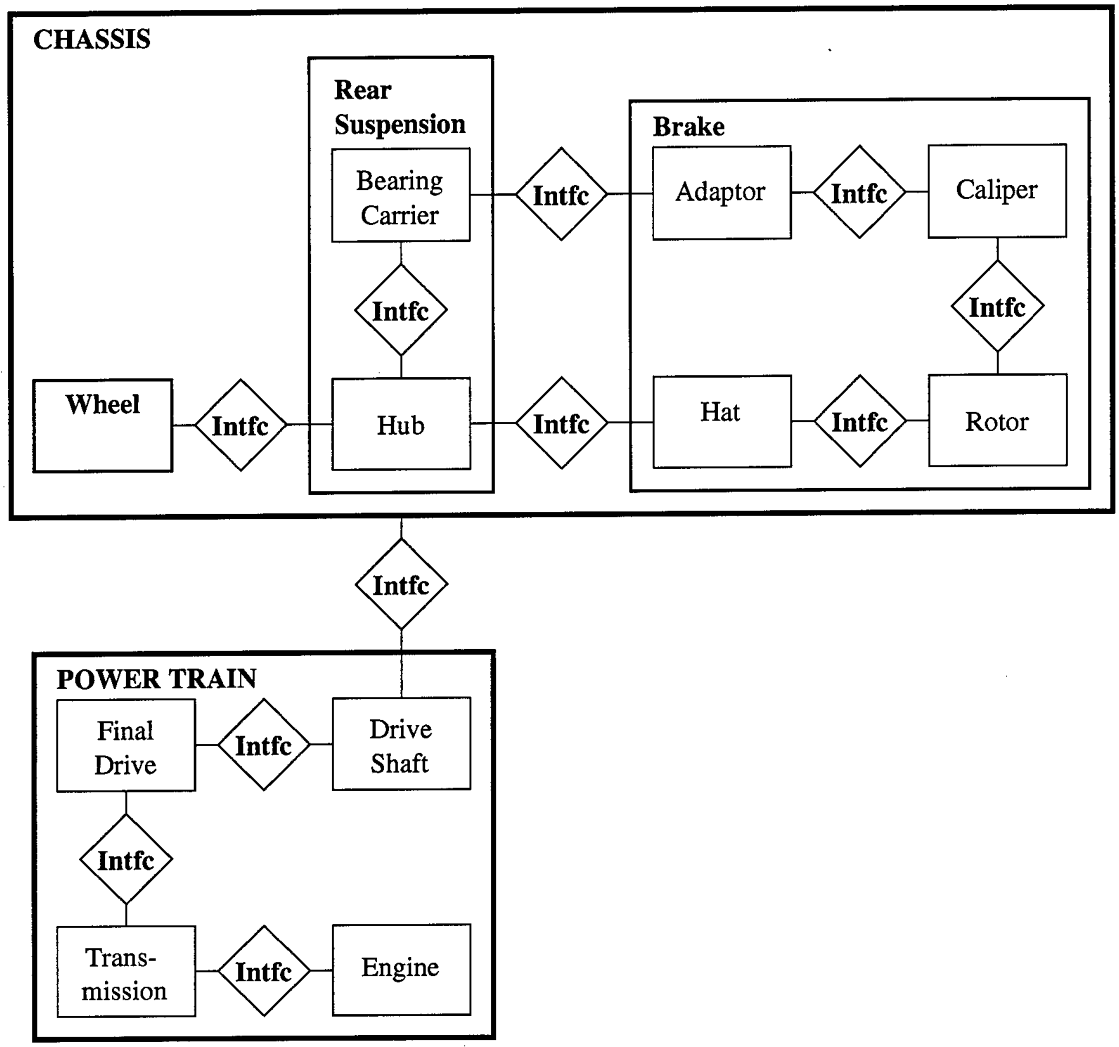



**Figure 4.11.** Structure of the formula automobile assembly

level geometry or parameters in a part can be grouped into a higher-level abstraction with a *neighborhood* aggregation. The hierarchy formed by the various aggregation objects organizes the model into abstractions that are easier to understand and manipulate. In addition, the *attachments* and *interface specification objects* in the part and assembly aggregations contain the information necessary for automatic positioning and validation of the design model.

# CHAPTER 5

# INTERACTION

The interactions between components in an assembly represent considerable complexity and risk in a product design. According to Sodhi and Turner [51], the relationships associated with these interactions are essential for depicting product functionality, since functionality can not be completely implemented solely within individual parts. In this research, an *interaction relationship* is defined as the collection of all significant information that specifies the cooperative behaviors between two components in an assembly. This information includes design functionality, force transmission, positioning and relative movement of parts, and fasteners and connectors. Previous research has tended to focus on individual aspects of the interaction relationship such as the orientation and relative motion of assembly components [16, 37, 49], the transmission of forces [27], or the inclusion of fasteners and connectors [1, 46].

It is through its interaction relationships that an assembly becomes more than the collection of its individual parts. The relative behaviors combine in synergistic ways, so it is insufficient to specify interaction information within the individual parts or even through hierarchy relationships of an assembly, as is done in many existing tools. Within a bolted joint, for example, the size of the fasteners and the thickness of the joint are key parameters in determining the load bearing capacity of the joint. Neither of these parameters, however, is unique to a single part. Since the joint thickness is derived from the interacting parts, it is dependent on a peer-to-peer relationship between the parts rather than the hierarchical relationships of the parts or assembly. As a means of allowing designers to more reliably specify and predict interaction relations, this research designs and implements the *interface*

*specification object.*

The interface specification object is a peer-to-peer relationship between two parts, components, or subassemblies. The peer-to-peer relationship is formed as a direct link between the two interacting parts or as a link between two *assembly features.* The assembly features are then embedded in the aggregation hierarchy of the interacting parts to complete the relationship.

The designer uses the interface specification object to specify and control behaviors in an interaction relationship. Within the interface specification object, kinematic behavior is constrained by a *joint* that describes the relative motion between the interacting parts. Other behaviors are specified and controlled by incorporating connectors, fasteners, forces, and other design information into the interface specification object.

Interface specification objects share the hierarchical attributes of part aggregation objects so that they can evolve along with the rest of the design. Thus, the interface specification object can form a hierarchy that collects all information relevant to the interaction between two parts or subassemblies. Since designers declare the content of these objects, they may leave out pertinent information, either because they are unaware of it, or because they deliberately choose not to control that aspect.

## 5.1 Role in Complexity Management

Specifying and controlling part interaction in interface specification objects supports management of design complexity by:

- Providing a means for representing design functionality. Functionality is generally manifested in multiple interacting parts. It cannot be adequately represented in individual parts since many of the functional parameters can not be attributed to any single part.

- Making available the information necessary for interactive analysis and simulation of moving parts. This is accomplished by incorporating kinematic

information into the interaction specification.

- Facilitating assemblability analysis, analysis of forces, or other design analyses by providing focal points in which to specify the appropriate interaction constraints.

- Encapsulating details, such as fasteners and connectors, thus eliminating the need for separate specification of these details in each independent part.

- Controlling the evolution of the individual part or subassembly design and reducing design incompatibilities. This can be accomplished either by using constraints from within an interface specification object to partially define related parts, or by ensuring the independently designed parts do not violate the constraints specified in the interface specification.

- Minimizing design complexity and the effect of changes. Behavior can be localized to one side of the interface specification, and change propagation can be controlled across the interface specification object, and hence across the interaction relationship that it embodies.

- Modularizing the design to facilitate independent development of related components and to simplify integration of these components into the remainder of the model. When coupled with the restricted access of aggregation objects, the interface specification object isolates interacting components from the remainder of the model. Thus, designers can independently develop a component in a local scope that does not affect the remainder of the model. If the component description adheres to the dictates of the interface specification objects, then it should be compatible when integrated with the remainder of the model.

- Facilitating design reuse of individual parts or subassemblies. This is done by separating the interaction relationship constraints from the part specification.

- Enabling the specification of cross-disciplinary information such as that involved in the interaction of electronic and mechanical subassemblies. This is

accomplished by incorporating the information into the aggregation hierarchy of the interface specification object.

- Enhancing the representation of product structure by adding peer-to-peer links between interacting parts. With peer-to-peer links, the designer can more easily analyze the impact one component has on another.
- Providing a common location in which distances between interacting parts can be specified. Such distances could include dimensional tolerances or parameters for building exploded views of an assembly.

## 5.2 Underlying Concepts and Terminology

The mechanisms presented in this chapter make use of specialized design objects for representing particular types of design information. The concepts and terminology associated with these specialized objects are described here.

### 5.2.1 Features

A feature encapsulates application specific design information into a reusable, standardized component that is mappable to a generic shape. Different features may be used to represent the same portion of a design, with each set of features capturing the design information that is relevant to a particular design discipline.

Similar to Shah and Mäntylä [49], this research uses the term *feature* to mean any design object with engineering significance. This may be a particular geometric shape, a previously defined feature in the design system, or an abstraction representing the combination of a number of different features or parts.

The *Alpha_1* system already includes a significant number of mechanical features. These features, when used in parts and interface specifications in the design model, include manufacturing information from which numerical control code can be generated for machining the part. Using the *Alpha_1* development environment, additional features may also be defined and incorporated into the complexity management framework from this research.

### 5.2.2 Joints

A key aspect of the interaction between parts in an assembly is the relative motion between these parts. This research utilizes *joints* to capture the degrees of freedom of motion and to enable the designer to interactively manipulate the motion of the two parts. As defined in this research, a joint is similar to the mating features used by other researchers (See Chapter 2); however, a joint extends the capabilities of mating features by allowing interactive manipulation.

This research defines a generic joint for conceptually linking two parts or assemblies before interaction details are described. In addition, *revolute, prismatic, spherical, against*, and *rigid* joint types are defined for constraining the allowable motion of interacting parts. A revolute joint allows rotation around a single axis and a prismatic joint allows translation in a single direction. A spherical joint allows three rotational degrees of freedom around a point, and an against joint allows two translational and one rotational degree of freedom on a surface. A rigid joint allows no motion between the two parts. *Alpha_1* also supports user-defined joint types in which the designer specifies the appropriate degrees of freedom.

The designer creates a joint by specifying the amount of rotational and translational movement for a particular joint type along with the current relative position of the two linked parts. The parts may be moved interactively by modifying the relative position in the joint. The joint object contains software assistants, that are automatically invoked when a joint is created or updated, to ensure invalid movements and positions do not occur.

### 5.2.3 Connectors

Gui and Mäntylä [27] introduce *connectors* to represent standardized components such as springs and gears that transmit energy between components. Similarly, this research defines bearing and screw connectors to represent and analyze the transfer of forces between parts. Although the current implementation of connectors supports only force transmission, this concept could be extended to other disciplines, such as heat or electricity transfer, with the development of additional connectors.

The mechanical connectors presented here encapsulate fasteners such as bearings

and screws into a single design object. This design object also encapsulates application parameters, geometry, and force analysis routines. The bearing connector includes one or more bearings and spacers that are coaxially aligned immediately adjacent to each other. Bearing life and rotational speed are application dependent parameters that must be specified by the designer. Using the screw connector, the designer can describe patterns of one or more screws of the same size and type along with application parameters for joint thickness and thread length.

Using software assistants embedded in the connectors, designers can automatically analyze simple point forces acting on an assembly. Similarly, the designer can invoke constructor commands to automatically generate manufacturing features such as a bearing bore or threaded hole that are compatible with the connector.

In this research, electronic catalogs are defined to facilitate the creation of connectors containing standard bearings or screws. To select a standard component, the designer need only specify a catalog number. The electronic catalog then searches its database and retrieves the geometry, force capacity, and other information for the selected component.

## 5.3 Interface Specification Object

This research introduces the *interface specification object* with which the designer can capture all design information relevant to the interaction between parts. The interface specification object is composed of assembly features and positioning constraints for the two interacting components and a joint describing the relative motion between the assembly features. The designer can incorporate additional information or levels of detail into the interface through an aggregation hierarchy similar to that of the part object discussed in Section 4.4.2. The structure of the interface specification object is shown in Figure 5.1.

Assembly features describe compatible features on each of the two interacting parts. Unlike other applications of assembly features, this research does not restrict assembly features to any particular design capability such as design for assembly or functional design. Instead, any functional, manufacturing, or form feature, or any

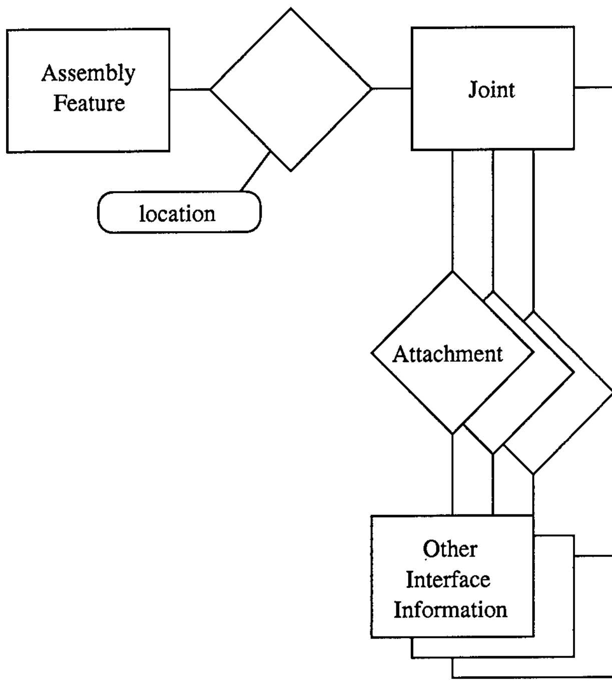



**Figure 5.1.** The interaction relation depicted in an interface specification object

geometric object available in the design system can serve as an assembly feature. The designer can also represent assembly features as aggregations with multiple levels of detail.

To ensure compatibility among interacting parts, the designer can incorporate assembly features from an interface specification object, that have been previously defined to be compatible, into a component model via the aggregation mechanisms of Chapter 4. When used in this way, any subsequent changes or additional detail added to the assembly features will cause the change propagation mechanisms of *Alpha_1* to automatically regenerate the related parts to include the changes. If the designer attempts a change that would lead to an invalid design model, the automatic regeneration stops and the designer is notified.

As additional detail is specified for a model, the designer adds this information to the interface specification by using *partof* relationships to link the details to the interface object aggregation hierarchy. While it is possible to incorporate any object in the design system into the aggregation hierarchy, some specialized aggregations are particularly applicable to the interaction between parts in an assembly. These

aggregations include:

- *Nested interface specifications* that can be used to reflect the decomposition of an interface into additional levels of detail.

- *Connectors* to support detailed force analysis across the interface and generation of manufacturing features such as threaded bolt holes.

- *Applied forces or force constraints* to provide the information necessary to carry out preliminary and detailed force analysis.

- *Additional constraints and analysis information* including information for interpretation by other tools.

The interface specification object provides the capability to describe many aspects of the interaction between parts in an assembly. It also provides a convenient mechanism for organizing and representing information used by other computational tools for analyzing the design model. The designer can maintain design compatibility by incorporating interface assembly features into the design of new part models or, when used with predefined part models, the designer can use the interface specification object to verify that the parts are compatible.

Interface specification objects may evolve along with the remainder of the design to represent functional concepts or detailed manufacturing information. Early in the design, the designer may only be interested in linking two components in a conceptual diagram. As the design evolves, the designer adds detailed information to the interface object to specify motion constraints, force analysis constraints, nested interfaces, fasteners and connectors, or other interaction information. Using the interface specification object and its associated aggregation relationships, the designer can interactively move parts in an assembly, analyze and modify common design parameters, and reconfigure assemblies to represent multiple levels of detail. The interface specification is an independent design object that the designer can manipulate to represent and control the complex relationships between interacting parts in an assembly.

## 5.4 Spindle Cartridge Subassembly

To demonstrate the analysis and control capabilities supported through the interface specification object, this section presents a scenario for the incremental design of the spindle cartridge subassembly from the machining center case study described in Section 3.2. The interactions between parts are essential to the functionality of the spindle cartridge subassembly and, consequently, the designer derives the individual components directly from these interactions. The innovative nature of the spindle cartridge design, as depicted in this scenario, benefits from an incremental design process in which the designer can carefully control and analyze incremental changes to the design.

The spindle cartridge holds and rotates cutting tools during the milling of a part. The spindle cartridge interacts with the spindle drive and the head casting to form the spindle head subassembly of the machining center as shown in Figure 3.5. The spindle drive provides power to spin the spindle cartridge and the head casting provides a base upon which the spindle cartridge is mounted.

In the first step of the incremental design process the designer identifies the operating environment and the external constraints imposed on the design. The spindle cartridge design is constrained by the size of the tools, the required milling accuracy, and the desired cutting speed. Although not part of the spindle cartridge, the tool holder holds cutting tools that interact with the part being milled, thereby exerting forces on the spindle cartridge subassembly. These forces, along with the interaction of the spindle cartridge with the head casting and the spindle drive, must all be considered in the design of the spindle cartridge.

To accommodate these external constraints, the designer uses the interface specification object to describe the interaction between the spindle cartridge and the tool holder, head casting, and spindle drive. In this scenario, the designer initially wants to analyze only the relative motion and forces acting on the spindle cartridge, so these constraints are added to the interface specification objects along with assembly features identifying the high-level geometry and manufacturing features associated with the interaction.

As shown in Figure 5.2, which describes the interface between the tool holder and the spindle, the designer uses an *intfcSeq* constructor to define the interface specification object. The *intfcSeq* constructor is an extension of the *partSeq* constructor that creates an aggregation containing a joint specification and two assembly features in addition to the information available in the part aggregation. In the interface between the tool holder and the spindle the designer specifies a *rigid* joint and two assembly features, labeled *toolholder* and *toolholder_slot*, to accommodate the tool holder part. Using the *intfcpos* and *intfcneg* constructors, the designer positions both the *toolholder* and *toolholder_slot* features at the base of the interface specification. The designer links two externally applied forces, *axialForce* and *radialForce*, into the interface specification hierarchy by establishing *partof* relationships with the joint.

```
toolholder_spindle_intfc : intfcSeq {

    "The interface between the toolholder and the spindle";

    "Joint allows no movement between toolholder and spindle";

    joint : rigid();

    "Select geometry from standard tool holder";

    toolholder : toolholderTaper40;
    toolholderSlot : reverseObj( toolholderTaper40 );

    "Identify and position positive and negative features";

    pos : intfcpos( baseAnchor, toolholder );
    neg : intfcneg( baseAnchor, toolholderSlot );

    "Attach forces acting on tool holder";

    atch1 : partof( joint, baseAnchor, axialForce );
    atch2 : partof( joint, XAnchor, radialForce );
};
```

**Figure 5.2**. Interface specification between tool holder and spindle

In the second step of the incremental design process, the designer decomposes the spindle cartridge into its major components: a spindle that rotates at a high rate of speed, a housing to provide a stable mounting for the spindle, and a drawbar for mounting the tool holder. The designer creates these components as conceptual entities by including them in the spindle cartridge assembly aggregation as shown in Figure 5.3. As the spindle cartridge is decomposed, the designer also identifies potential interactions between the spindle, the housing, and the draw bar. These interactions are represented initially by interface specification objects with generic joints (*ijoint*) linking the conceptual entities (*spindle, housing,* and *drawbar* identified in the assembly. The designer describes design goals, such as desired *fatigue life* and *speed*, by listing them as variables in the assembly aggregation.

In this scenario, part interactions are critical to the functionality of the spindle cartridge subassembly, so the designer wants to ensure parts are compatible with the interface specification objects linking them. To facilitate the design of compatible

```
spindleCartridge : assemblySeq {

    "Performance parameters and goals";

    FatigueLife :* 10000;
    CuttingSpeed :* 4000;

    "Components in spindle cartridge subassembly";

    spindle;
    housing;
    drawbar;

    "Interfaces between components in subassembly";

    spindle_housing_Intfc :
        intfc( ijoint(), baseAnchor, spindle, baseAnchor, housing );
    spindle_drawbar_Intfc :
        intfc( ijoint(), baseAnchor, spindle, baseAnchor, drawbar );
};
```

**Figure 5.3**. Specification of spindle cartridge subassembly

parts, the designer intends to build the interface specification objects and then derive part models from these specifications. Since the spindle is a shaft that spins within the housing, the designer constructs an interface specification object containing a revolute joint with unlimited rotation, a through-hole feature, and a cylindrical shaft feature as depicted in Figure 5.4(a). The geometry of the spindle-housing interface, as shown in Figure 5.4(b), represents the compatible assembly features that are described in the interface specification object.

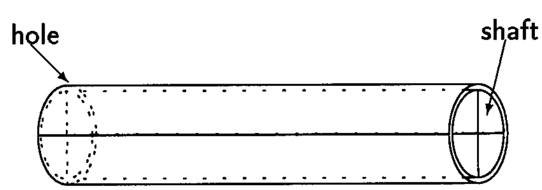
To describe the interaction between the draw bar and the spindle, in which the draw bar shaft moves in and out of the spindle like a piston, the designer creates another interface specification object. The spindle-drawbar interface specification includes a prismatic joint with limited movement allowed along the axis, a hole feature, and a cylindrical shaft feature.

The designer proceeds with the incremental design process by describing the high-level design of the major components within the spindle cartridge subassembly. To model the spindle, the designer creates a part aggregation, as shown in Figure 5.5, into which he inserts the negative feature of the tool holder-spindle interface (*toolholder_spindle_intfc.negEntity*) and the hole feature of the spindle-drawbar interface (*spindle_drawbar_intfc.negEntity*). The designer attaches these two features with *partof* relations to the shaft feature of the spindle-housing interface (*spindle_housing_intfc.posEntity*). The *part* aggregation constructor automatically aligns the attached parts and incorporates them into a part model for the spindle. The designer creates the housing and drawbar parts in a similar fashion. By deriving parts from the interface specifications as shown here, any changes the designer makes to the interface specification objects will be automatically reflected in the part model, thus maintaining compatibility between the interacting parts.

As the designer adds additional detail to the interface specification objects and part aggregations, *Alpha_1* automatically propagates these changes to the spindle cartridge subassembly specified in Figure 5.3. The constructor function for the assembly aggregation uses the spindle part as a base part and automatically aligns the remaining parts according to their positions in the interface specification

```
spindle_housing_intfc :* intfcseq {

    "Revolute joint with complete rotation";

    joint : revolute( unlimited );

    "Shaft is a cylindrical surface of revolution";

    shaft :* surfrev( profile( pt( shaft_radius, 0, shaft_height ),
                               pt( shaft_radius, 0, 0 ) ),
                      true );

    "Hole is a mechanical hole feature";

    hole :* entity( hole( offsetanchor(baseAnchor, 0.02, 0.02, 0.02),
                          shaft_radius * 2 + 0.3,
                          shaft_height,
                          0,
                          TRUE ) );

    "Shaft is positive side of interface and hole is negative side";

    pos :* intfcpos( baseAnchor, shaft );
    neg :* intfcneg( baseAnchor, hole );
};
```

(a) Interface specification




(b) Feature geometry

**Figure 5.4.** Initial interface specification between spindle and housing parts

```
spindle : partseq {

    "The spindle is defined by the shaft side of the spindle-housing
     interface, the hole side of the spindle-drawbar interface, and
     the hole side of the toolholder-spindle interface.";

    "Extract features from the appropriate interfaces";

    shaft : spindle_housing_intfc.posEntity;
    hole : spindle_drawbar_intfc.negEntity;
    tool : toolholder_spindle_intfc.negEntity;

    "The shaft feature is the main geometry of the spindle";
    "The hole features are attached to the shaft";

    atch1 : partof( shaft,
                    offsetAnchor( baseAnchor, 0, 0,
                                  -spindle_toolholder_offset ),
                    tool,
                    "-" );
    atch2 : partof( shaft,
                    offsetAnchor( baseAnchor, 0, 0,
                                  spindle_drawbar_offset ),
                    hole, "-" );
};
```

**Figure 5.5**. Spindle part specification

objects. Figure 5.6 shows the resulting geometry of the initial spindle cartridge subassembly.

Figure 5.7 shows the logical model structure for the assembly aggregation containing the spindle and housing parts linked with the spindle-housing interface specification object. This diagram depicts each component as an aggregation with attachments (*atch*) showing the hierarchical relationships. The *hole* and *shaft* features defined in the spindle-housing interface specification object are linked into the housing and spindle parts with attachment relationships to reflect the incorporation of these interface features into the actual design model of the individual parts.

Now that the initial design increment of the spindle cartridge is complete, the

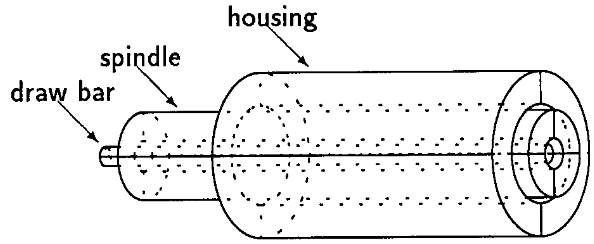



**Figure 5.6.** Initial spindle cartridge subassembly

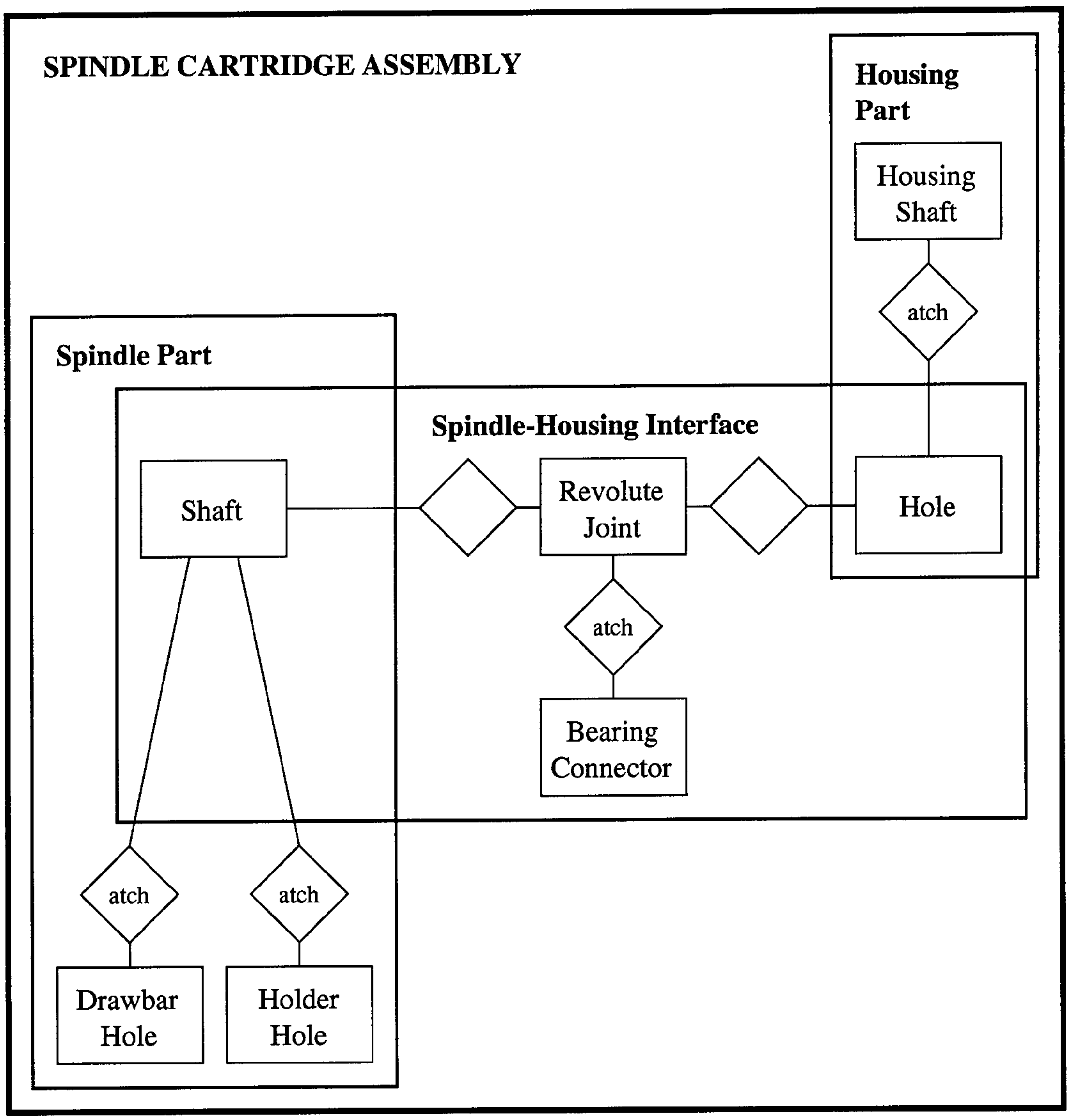



**Figure 5.7.** Structure of spindle-housing assembly

designer analyzes the design model to validate assumptions and constraints, to evaluate the satisfaction of design goals, and to determine how to proceed with the next design increment. With just a rough description of the part geometry and the interfaces, the designer invokes automated software assistants, that focus on the aggregation and interaction relationships, to analyze the forces acting on the entire assembly. The designer accomplishes this with the command:

```
validateForces( SpindleCartridge )
```

The assembly aggregation constructor automatically validates the position of interface joints when the assembly is updated. The designer can manipulate joint positions or key parameters upon which the joints are dependent to analyze the kinematic behavior of the subassembly. If problems are discovered, the designer can concentrate analysis on individual interface specification objects to isolate the problems. The designer refines constraints, parameters, and design components as necessary, and *Alpha_1* automatically regenerates component models, until all problems are resolved.

When satisfied with the results of the first design increment, the designer refines the design by adding additional detail. The bearings between the spindle and the housing are key components in determining milling speed and fatigue life. Using the *lookupbearing* command, the designer provides an identification number to automatically retrieve parametric models of bearings with the proper dimension and estimated force capacity from an electronic catalog as shown in Figure 5.8(a). The *bearingconn* constructor in this figure allows the designer to create a bearing connector containing three bearings and a spacer along with parameters specifying desired fatigue life and rotational speed. To facilitate the modification of existing aggregations, this research provides the *merge* command. In Figure 5.8(a), the designer uses this command to incorporate the connector into the spindle-housing interface aggregation. Associated with the bearing connector are additional constructors, *bearingConnStep* and *bearingConnBore*, that the designer uses to generate manufacturing features, on both the shaft and the housing, to

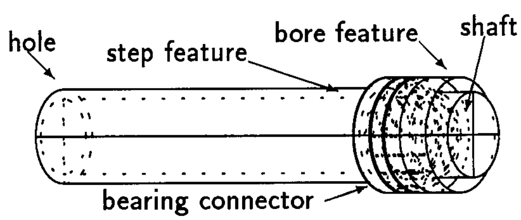
accommodate the bearing connector. Figure 5.8(b) shows the new assembly features generated after adding the *step* feature to the shaft and the *bore* feature to the housing. After the designer merges the bearing details into the spindle-housing interface, *Alpha_1* automatically regenerates the housing and spindle parts with the appropriate bearing connection features and propagates all changes to the entire spindle cartridge subassembly.

So that the bearings can be inserted during the assembly process and held in place during operation, the designer decomposes the housing part into another subassembly containing the main housing part and a detachable nose cap at the end of the housing as shown in Figure 5.9. This assembly requires a new interface between the nose cap and the housing. The designer determines that the nose cap is to be held in place with screws, so the designer invokes a command to retrieve the desired screw from an electronic catalog, then creates a screw connector using a constructor that arranges six identical screws in a radial pattern. The designer attaches the screw connector to the nosecap-housing interface using a *partof* relationship, and builds assembly feature aggregations that include the screw features. The designer constructs a new aggregation for the nosecap part in which he attaches the assembly features of the nosecap-housing and spindle-housing interfaces. The designer also attaches the nosecap-housing interface to the housing part, causing *Alpha_1* to automatically update the model of the housing part to include the threaded hole features for the screw connector. After the individual parts and interfaces are updated, *Alpha_1* automatically regenerates the spindle cartridge subassembly with the nosecap-housing interface and the new parts added. The updated geometry is shown in Figure 5.10.

With the completion of another design increment, the designer now needs to confirm that the proper bearings and screws were used. Each connector has characteristics, such as bearing life or screw grade, that determine the force limits that the connector can withstand. The designer invokes software assistants for force analysis that automatically calculate these limits and notify the designer if the applied forces exceed the capabilities of the connectors.

```
spindle_housing_intfc :* merge {
  "Retrieve bearing from electronic catalog; Insert 3 bearings and
   spacer into connector; specify fatigue life and speed goals in
   connector; and attach connector to interface joint";
    bearing : lookupbearing( "L13", bearingCat );
    spacer : bearingSpacer( bearing, 14 * mmtoinch );
    bearingconn : bearingconn(array(bearing, bearing,
                                    spacer, bearingInvert(bearing)),
                               SpindleCartridge::FatigueLife,
                               SpindleCartridge::Speed);
    bearingAnch : offsetanchor( baseAnchor, 0, 0, bearing_offset );
    atch1 : partof( joint, bearingAnch, bearingconn, "+" );
  "Generate step feature from bearing connector and attach step
   feature to interface shaft.  Make this the positive feature.";
    shaft_part :* partSeq {
        shaft : shaft;
        shaftStep : bearingConnStep( bearingconn, step_length, 0 );
        atch1 : partof( shaft, bearingAnch, shaftStep, "-" );
    }
    pos :* intfcpos( baseAnchor, shaft_part );
  "Generate bore feature from bearing connector and attach bore
   feature to interface hole.  Make this the negative feature.";
    hole_part :* partSeq {
        housinghole : hole;
        housingbore : bearingConnHole( bearingconn, bore_length, 0 );
        atch1 : partof( housinghole, bearingAnch, housingbore, "+" );
    }
    neg :* intfcneg( baseAnchor, hole_part );
};
```

(a) Interface specification




(b) Feature geometry

**Figure 5.8.** Details of the spindle-housing interface

```
housingAssy : assemblySeq {

    "Decompose the housing into an assembly";

    housing;
    nosecap;
    nosecap_housing_intfc;
};
```

**Figure 5.9.** Housing subassembly specification

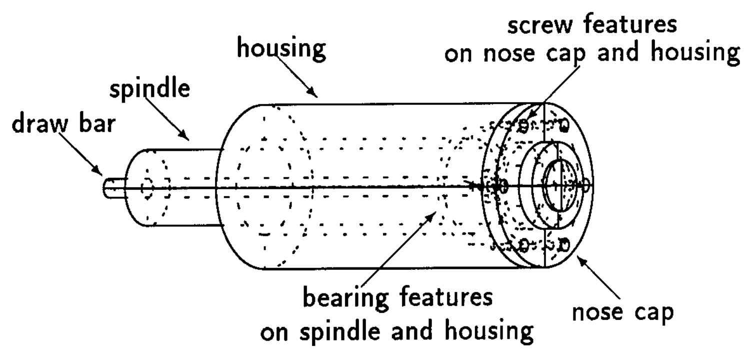



**Figure 5.10.** Detailed spindle cartridge subassembly

At this point in the design process, the designer has encapsulated descriptions of functionality and design rationale; design parameters, constraints, and goals; forces and kinematic information; manufacturing features; and geometry into the interface specification objects of the spindle cartridge subassembly. From this information the designer can invoke automated procedures, provided through this research or existing capabilities in *Alpha_1*, to analyze the forces and kinematic behavior of the subassembly, calculate geometric interference, or generate manufacturing process plans. The designer can generate different alternatives for the components or interfaces and *Alpha_1* automatically propagates these changes to the affected parts and subassemblies. The designer has also decomposed the machining center design problem into small subassemblies that are more easily understood and managed, and he has restricted the design of the spindle cartridge subassembly through external interface specifications that ensure its compatibility with the remainder

of the machining center.

## 5.5 Formula Automobile Examples

The formula automobile case study, with its focus on simultaneous design and higher-level subassemblies, demonstrates additional capabilities of the interface specification object. At higher levels, the automobile is decomposed into multiple subassemblies that are independently designed by separate subteams to satisfy predefined specifications. At lower levels, design subteams develop complex new designs or modify and reuse existing designs. Each independently designed subassembly is eventually integrated into a higher-level assembly until a completely integrated product design is achieved.

### 5.5.1 Evolution of the Brake - Suspension Interface

A considerable portion of the design of a part or subassembly is determined by its interaction with other components. The brake subassembly of the formula automobile, for example, is largely defined by its interactions with the rear suspension subassembly. This is demonstrated in the specification and evolution of the interface between the brake and rear suspension subassemblies.

When the interaction relationship between the brake and the rear suspension assemblies is first identified, the details are not well-defined. At this point, the design team defines an interface specification object to serve as a structural link between the two subassemblies. This link is generated with the constructor:

```
intfc( ijoint(), brakePosn, Brake, suspPosn, RearSuspension )
```

that creates a link between the two subassemblies. This link contains a generic joint and allows the *Brake* and the *RearSuspension* to be positioned according to the *brakePosn* and *suspPosn* anchors. Geometry and position information associated with the interface at this point is useful for generating structural diagrams as shown in Figure 5.11.

As the brake and rear suspension subassemblies are decomposed into their separate components, the design subteams realize that the interface between the two

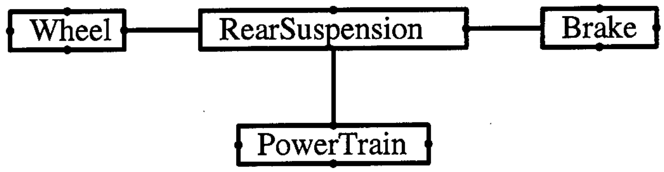



**Figure 5.11.** Diagram of rear layout components

subassemblies must also be decomposed. One part of the brake subassembly rotates with the hub and the wheel, while another part must be fixed to the rear suspension to stop the rotation. In accordance with the decomposition from Section 3.1, the designers create additional interfaces between the brake adaptor and the bearing carrier and between the brake hat and the hub. The designers incorporate these additional interfaces into the interface between the brake and the rear suspension using *partof* relations as shown in Figure 5.12. At the same time, the designers identify some common dimensions, labeled as *CalMountOffset* and *BrakeEarOffset* in Figure 5.12, that affect both of the subassemblies.

The interfaces between the brake and the rear suspension continue to evolve as the designers add more detail to the interacting subassemblies. During this evolution, the designers first describe the interaction information in the interface specification object and they then incorporate this information into the interacting components. The interfaces between the brake and rear suspension subassemblies demonstrate useful capabilities for representing and controlling design evolution using the interface specification object.

The interface between the brake adaptor and the bearing carrier consists primarily of a rigid, bolted connection as shown in Figure 5.13. What makes this interface specification so useful, however, is the location and dimension parameters that the designer includes in the interface specification object. The designer uses these parameters in the design of the brake adaptor and the bearing carrier to ensure the two parts are compatible.

The brake hat and the wheel interact with the hub of the rear suspension

```
Brake_RearSuspension_Intfc : intfcSeq {

    "Include generic link information";

    joint : ijoint();

    pos : intfcPos( baseAnchor, Brake );
    neg : intfcPos( baseAnchor, RearSuspension );

    "Identify common dimensions";

    CalMountOffset :* ( 37.5 );
    BrakeEarOffset :* ( 40.0 );

    AdaptorAnchor : offsetAnchor( baseAnchor,
                                  CalMountOffset,
                                  BrakeEarOffset,
                                  0 );

    "Decompose into interfaces between Adaptor and Bearing Carrier
     and between Brake Hat and Hub";

    Adaptor_BearingCarrier : intfc( ijoint(),
                                    baseAnchor,
                                    Adaptor,
                                    baseAnchor,
                                    BearingCarrier );
    atch1 : partof( joint, AdaptorAnchor, Adaptor_BearingCarrier );

    BrakeHat_Hub : intfc ( ijoint(),
                           baseAnchor,
                           BrakeHat,
                           baseAnchor,
                           BearingCarrier );
    atch2 : partof( joint, baseAnchor, BrakeHat_Hub );
};
```

**Figure 5.12.** Decomposition of interface between brake and rear suspension

```
Adaptor_Carrier_Intfc : intfcSeq {
  "Location and Dimensional Parameters";
    CalBoltSep :* ( 80.0 );
    CalBoltOffsetY :* ( BrakeEarOffset - CalBoltSep );
    CalBoltOffsetX :* ( CalMountOffset + 30.0 );
    Offset1 :* ( 7.25 );
    Offset2 :* ( 8.0 );
    BrakeRotorThick :* ( 4.75 );
    BrakePadThick :* ( 13.0 );
    BrakePadAllow :* ( BrakePadThick + 0.5 );
    BrakeCalFlangeRad :* ( 8.0 );
    BrakeCalFlangeThk :* ( 8.0 );
    BoltHoleDia :* ( 8.0 );
    AdaptorThk :* ( 6.5 );
    CarrierThk :* ( 10.0 );
  "Interface joint and surfaces";
    joint : rigid();
    surf : capSurface(
        profile( pt( CalMountOffset .... BrakeEarOffset ... ) ) );
  "Retrieve screw from electronic catalog; build screw connector;
   and attach connector to interface joint";
    screw : lookupScrew( "11F", screwThreadTable,
                         "SOCKET", screwHeadTable,
                         2, screwGradeTable,
                         AdaptorThk + CarrierThk, AdaptorThk );
    screws : screwrect( screw, AdaptorThk, 2, CalBoltSep, 1, 0 );
    screwAnchor : offsetAnchor( baseAnchor, 0, 0, -AdaptorThk );
    atch1 : partof( joint, screwAnchor, screws, "+" );
  "Generate screw features and attach to interface surfaces";
    adaptorPart : partSeq {
        surf : entity( surf );
        atch1 : partof( surf, screwAnchor, screws, "+" );
    }
    pos : intfcpos( baseAnchor, adaptorPart );
    carrierPart : partSeq {
        surf : entity( reverseObj( surf ) );
        atch1 : partof( surf, screwAnchor, screws, "+" );
    }
    neg : intfcneg( baseAnchor, carrierPart );
};
```

**Figure 5.13.** Interface between brake adaptor and bearing carrier

subassembly through a rigid, bolted connection. Each of these three components interacts with the others in a manner that appears to require three separate interfaces, yet a single bolted connection is used to join all three parts. The designers overcome this dilemma by deriving the interface between the brake hat and the hub from the previously defined interface between the wheel and the hub as shown in Figure 5.14. In this derivation, the designer creates a separate interface between the brake hat and the hub by copying the hub-wheel interface. The only thing the designer changes in the derived interface is the actual positioning of the interface bolt connector. Consistency among the derived interfaces is maintained through the change propagation mechanisms of *Alpha_1*.

### 5.5.2 Reuse of the Wheels

In the formula automobile case study, the design team wants to reuse a previously designed model of the wheel assembly. In doing so, the design team must be able to ensure compatibility with the remainder of the design. The designer uses the interface specification object between the hub and the wheel to help achieve this compatibility. If the wheel is compatible with the interface specification object,

```
"Derive the brakehat-hub interface from the hub-wheel interface";

BrakeHat_Hub_Intfc :* Hub_Wheel_Intfc;

BrakeHat_Hub_Intfc :* merge {

    "Offset the interface to account for the thickness of the
     brake hat";

    BoltAnchor : offsetAnchor( baseAnchor, 0, 0, -HubThk - 2 * Ext );
    atch1 : partof( joint,
                    BoltAnchor,
                    HubBoltHolePattern,
                    "+" );
};
```

**Figure 5.14.** Derivation of interface between brake hat and hub

then it should be compatible with the remainder of the model.

The designers integrate the wheel into the design model by transforming the wheel model into the current design space, aligning it with the proper side of the interface, and invoking automated software assistants to check compatibility. In this example, the wheel is designed in inches and is offset from the origin. The local design space requires metric dimensions and orientation aligned with the Z axis and centered at the origin. The transformation to the local design space is accomplished with the command:

```
Wheel : entity( objTransform( WheelOriginal,
                              array( tz( WheelOffset ),
                                     sg( metricConv ) ) ) );
```

The designer aligns the wheel with the interface between the hub and the wheel by incorporating the wheel into a part that has the negative feature of the interface specification object attached through a *partof* relationship:

```
WheelPart : part( Wheel,
                  partof( Wheel, Posn,
                  Hub_Wheel_Intfc.NegEntity ) );
```

Finally, to calculate geometric compatibility, the designer invokes a command to check if there is any interference in the newly defined wheel part:

```
checkInterference( WheelPart );
```

Now the designer can control and manipulate the wheel part like any other component in the assembly. Any changes the designer makes to the original wheel will cause *Alpha_1* to automatically reflect the changes in the new wheel part; however, after any change the designer will still need to recheck compatibility.

### 5.5.3 Kinematics of the Drive Shaft Interfaces

Although the assembly constructor automatically invokes kinematic analysis for any interface containing a kinematic joint, this analysis has been difficult to visualize on previous examples because they involved revolute surfaces and revolute joints. As a better visualization of kinematic analysis capabilities, this example analyzes

the rear layout assembly of the formula automobile with respect to the vertical travel of the wheel that might result from hitting a bump in the road. To simplify the analysis, the rear suspension support members and springs are not included. Instead, the designer focuses the analysis on the movement of the drive shaft that interacts with the final drive as shown in the simplified interface specification of Figure 5.15. In this interface the designer uses a spherical joint to allow limited rotation around the $X$, $Y$, and $Z$ axes according to the first three parameters of the *spherical* constructor. The designer describes the rotational position about the $X$ axis, the fourth parameter of the constructor, in terms of wheel travel (*WheelPosn*), which is an independent variable of the analysis. The positive and negative features are points used by the designer simply to connect the interacting parts. The designer specifies the interaction between the drive shaft and the hub in a similar way, and incorporates all of the subassemblies and interfaces of the rear layout assembly into an assembly aggregation as shown in Figure 5.16. Upon creation, the assembly constructor automatically analyzes the joint kinematics based on the rotational positions about each axis. In this example, rotation about the $X$ axis is described in terms of wheel travel, while rotation about the $Y$ and $Z$ axes defaults to the center of the allowable range of rotation. By varying the wheel travel parameter, the designer can update the assembly, which causes the assembly constructor to

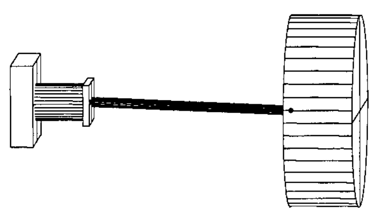
```
FinalDrive_Shaft_Intfc : intfcSeq

    "Spherical joint";

    joint : spherical( 15, 5, 2, atand( WheelPosn, ShaftLength ) );

    "No surface details necessary for kinematic analysis";

    pos : posIntfc( baseAnchor, pt( 0, 0, 0 ) );
    neg : negIntfc( baseAnchor, pt( 0, 0, 0 ) );
;
```

**Figure 5.15**. Kinematic interface specification between drive shaft and final drive

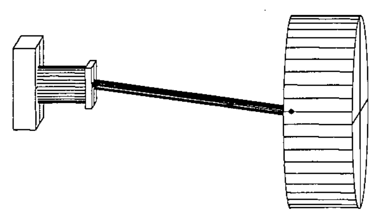
```
rearLayout : assemblySeq {

    "Rear layout of formula SAE automobile";

    "This layout groups all components connected
     through rigid interfaces into assemblies";

    FinalDrive;

    DriveShaft;

    BearingCarrierAssy : assemblySeq {

        BearingCarrier;
        BrakeAdaptor;

        Adaptor_Carrier_Intfc;
    };

    WheelAssy : assemblySeq {

        Rotor;
        BrakeHat;
        Hub;
        Wheel;

        Rotor_BrakeHat_Intfc;
        BrakeHat_Hub_Intfc;
        Hub_Wheel_Intfc;
    };

    FinalDrive_Shaft_Intfc;
    Hub_Shaft_Intfc;
    Hub_Carrier_Intfc;
};
```

**Figure 5.16.** Specification of rear layout assembly

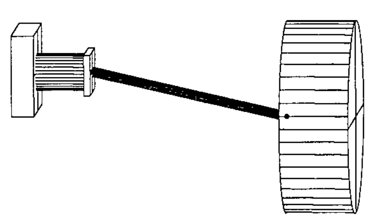
reaccomplish the kinematic analysis. If the wheel travel will result in an invalid position for the spherical joints, the assembly is not reconstructed and the designer is notified. Figure 5.17 shows the resulting assembly at different wheel positions.



(a) Wheel travel = 25mm



(b) Wheel travel = 75mm



(c) Wheel travel = 125mm

Invalid X position

(d) Wheel travel = 175mm

**Figure 5.17.** Rear layout assembly at different wheel positions

# CHAPTER 6

# VARIATION

Variation is an essential characteristic of any design process. As a design evolves from conceptual ideas to detailed features and geometry, designers create different variations of the design with increasing amounts of detail. If new requirements are imposed on the product, the design must be modified to accommodate these requirements. Designers often generate alternate variations of a design when exploring potential solutions to a design problem.

Variation management, or as it is more commonly referred to, version management, is the process of controlling changes to a design component and tracking the differences between versions of the design component once changes have been made. When different views or alternatives are involved, version management also ensures that the alternatives are kept consistent when designers make changes.

Most CAD environments provide little or no support for version management. Frequently, the designer can only view the current version of a design model with little information about the process and decisions through which the complete model came about. To manage multiple alternatives or old versions, the designer must save them under different names and track them independently from the actual design model. In systems that do provide support for managing design variations, the designer has little control over the level of granularity at which design models are tracked and controlled. Typically, it is convenient for a designer to manage only complete design models with these systems.

This research designs and implements automated mechanisms for creating and managing versions of assembly, part, and neighborhood aggregations. To track designer modifications of an aggregation, this research creates automated software assistants that compute and record the differences in a new version of the aggre-

gation. This research also includes software assistants that automatically generate versions representing alternative solutions or views of the design, and it implements simple commands for selecting or copying existing versions of a component model.

To simplify the process of changing a design model, this research introduces new editing mechanisms into the *Alpha_1* design environment. The *merge* operation modifies existing components or adds new components to an aggregation. A *long transaction* sets the scope being edited to that of a designated aggregation. Within a long transaction, designers modify an aggregation locally without impacting the remainder of the product model.

A goal of this research is to make version management an interactive design tool. To achieve this goal, this research creates versions based on the design aggregation objects described in Chapter 4. Through aggregation specifications, designers can control version granularity in a manner that is appropriate for their application or process. In addition, with the version selection operations of this research, designers can interactively access different versions of a design component for concurrent analysis, design exploration, or to manage design variants.

## 6.1 Role in Complexity Management

Any modification to a design object represents a new variation of that design object. Management of these variations, however, requires an acknowledgment of the intent of the modification. This research classifies variations into two categories, *refinements* and *alternatives*, based on the designers intent in creating the variation.

This research uses a refinement to portray modifications to a particular aspect of a product design model that do not change the primary functionality or basic design approach. If slight changes are made to the product requirements, designers must refine the design to adapt to these changes. If there are problems or deficiencies with the current design, the designer creates refinements to perfect or improve the design. As a design evolves from a high-level concept to a manufacturable model, the product model is refined by adding more detail. In the framework introduced by this research, more refined levels of detail are represented as new configurations

of an aggregation that contain additional, more detailed design components.

An alternative, as implemented in this research, describes an additional design solution for a particular aspect of a design problem or a separate view of a design solution to represent different design disciplines for analysis. Alternatives initially reflect a similar level of design detail; however, selected alternatives may evolve to include additional information.

By managing design variations, designers have added control over product model evolution. By creating new versions of a design representation instead of overwriting an existing version, designers can recover previous versions. Version recovery may be necessary so the designer can correct errors that were introduced subsequent to that version or so that the designer can explore an alternate variation without the detailed information of the current version. The variational tools of this research can be used by the designer to manipulate and experiment with a versioned component in isolation from the remainder of the product model. In this way, the designer can perfect a single segment of the model before propagating the results to the remainder of the model.

To explore the design space, a designer can use automated software assistants from this research to create alternative solutions to a particular design problem and to interactively select and analyze the alternatives to determine which is more appropriate in different scenarios. The designer can generate these alternatives from the same base component so that the alternatives share common constraints or features.

Designers can use the aggregation mechanisms discussed in Chapter 4 to include design rationale and descriptions in design models. This information becomes part of the variation and can be recovered and used by a different designer to further evolve a design or to modify and reuse a particular design component.

## 6.2 Underlying Concepts and Terminology

Version management in *Alpha_1* is dependent on two key data structures, *scope* and *model object*, that form the relationships necessary to depict the hierarchies and dependencies of a design model. Although all design objects require these

structures to become part of a model, the scope and model object structures are largely hidden from the designer.

### 6.2.1 Scope

The neighborhood, part, and assembly aggregations from Chapter 4 form design hierarchies. Many related design objects can be nested into a single aggregation object. Each object in the aggregation becomes a member of a *scope* that is associated with the aggregation object. The scope limits the accessibility of the nested objects by objects external to the aggregation.

### 6.2.2 Model Object

The *model object* identifies key information such as the constructor function and the prerequisite pointers necessary to create a design object. The precedence relationships of a design model are depicted in the prerequisite pointers of model objects and the associated pointers to dependent objects that are based on an object. Hierarchical relationships are represented with pointers to nested scopes that are embedded in an aggregation design object. Each model object is a member of a scope that is referenced through the model object.

## 6.3 Automated Mechanisms

This research presents automated versioning mechanisms that help the designer to manage and to control variation at different levels of detail. The designer can use these mechanisms to track revisions, generate alternatives, and check consistency between related views. The versioning mechanisms created for this research include automated routines for generating baseline versions, delta versions, or alternative versions. In addition, simple commands have been implemented for interactively selecting versions.

### 6.3.1 Baseline

This research defines a *baseline* as a complete version of a design aggregation at a particular instance in time. A complete version contains all of the components within the nested scope of a neighborhood, part, or assembly aggregation and the

prerequisites necessary to reconstruct the aggregation object. To implement the automated versioning mechanisms in this research, the model object constructor routines of *Alpha_1* were manipulated to make a copy of the old scope before allowing any modifications. Additional routines were added to *Alpha_1*, so when a designer invokes a command to update an aggregation, the newly updated version is automatically appended to a version list for that aggregation. A baseline version is created automatically as the original version of an object. By invoking the *baseline* command, the designer can also explicitly create a baseline version, thus creating a checkpoint to facilitate design recovery.

Figure 6.1(a) demonstrates the initial creation of a versioned aggregation object for the rear layout of the formula automobile. Version creation is completely automated – once the designer identifies an aggregation with the ":*" assignment operator, future updates to the aggregation cause a new version to be automatically created. In this example, the initial version includes all design objects defined within the brackets.

### 6.3.2 Delta

A *delta* version contains the differences between the current and previous versions. By including only the differences instead of the complete version, a delta version saves computer storage space. A delta version is automatically created when a designer updates an existing version.

This research provides three different mechanisms for a designer to revise aggregation objects:

- *Complete reconstruction.* A new aggregation constructor specifies all unchanged objects from the original version along with all new or modified objects. A new version is automatically created upon successful execution of the new constructor.

- *Merge operation.* A *merge* constructor specifies all new or modified design objects in the aggregation. The merge constructor creates a new version of the original aggregation that contains the new or modified objects.

- *Long transaction.* The local scope of the editor is changed to that of a designated aggregation. The designer can then interactively edit the contents of the aggregation by modifying or adding design objects. Any changes the designer makes during the long transaction affect only the scope of the aggregation object for which the transaction was invoked. The long transaction termination command automatically creates a new version containing the objects that were added or modified during the transaction. At this point, *Alpha_1* propagates the results of the long transaction to the rest of the model.

  The term *long transaction* is adapted from database theory [65] to describe an editing session that continues over an indefinite period of time. The designer controls when the long transaction begins and ends; however, in the current implementation, the designer must end a long transaction in the same editing session that it was started. The designer starts a long transaction with the *BeginScopeEdit* command that includes an argument identifying the aggregation to edit. For example, to edit the contents of the *RearLayout* assembly aggregation, the designer invokes the command:

  ```
  BeginScopeEdit( RearLayout );
  ```

  The designer ends a long transaction with the *EndScopeRevise* or the *EndScopeAlternate* commands. These commands create a revised or alternative version of the aggregation, commit the changes, and propagate the changes to the rest of the model. The *EndScope* command affects the last scope for which a *BeginScopeEdit* command was issued.

In Figure 6.1(b) the designer uses the *merge* constructor to create a new version of the *RearLayout* assembly aggregation created in Figure 6.1(a). These two versions demonstrate the utility of the variation mechanisms for representing various design abstractions. Each version represents a different level of detail for the same object, and the designer can access each version individually to view the model at either of the two levels of detail.

```
RearLayout :* assemblySeq {
  "Layout of rear section of automobile";
    RearSuspension;
    PowerTrain;
    Brake;
    Wheel;
    Suspension_PowerTrain_Intfc;
    Brake_Suspension_Intfc;
    Suspension_Wheel_Intfc;
}
```

(a) Initial version

```
RearLayout : merge {
  "Add detailed subassemblies for the
   brake, rear suspension, and powertrain";
    Brake : assemblySeq{
        Hat;
        Rotor;
        Caliper;
        Adaptor;
        Hat_Rotor_Intfc;
        Rotor_Caliper_Intfc;
        Caliper_Adaptor_Intfc;
    }
    RearSuspension : assemblySeq{
        BearingCarrier;
        Hub;
        Carrier_Hub_Intfc;
    }
    PowerTrain : assemblySeq{
        DriveShaft;
        FinalDrive;
        Engine;
        DriveShaft_FinalDrive_Intfc;
        FinalDrive_Engine_Intfc;
    }
}
```

(b) Revision with merge command

**Figure 6.1.** Versions of rear layout assembly

The examples in Figure 6.1 also include design descriptions and rationale in the versioned aggregations. This information becomes part of the version history and can be used by a designer to understand why certain decisions were made and how the design evolved to its current state.

### 6.3.3 Alternative

An *alternative* version depicts an additional design solution or view of an object. This research implements an alternative as a complete version of an aggregation object that starts a parallel version path. Once an alternative has been created, it can be revised just like any other version. Revisions to alternative versions to not affect other alternatives of the same object.

To create an alternative, designers have the same options – complete redefinition, merge, or long transaction – that are available to create a revision of an object. A designer distinguishes the construction of an alternative from a revision by the assignment operator "`:<`".

The versioning mechanisms presented in this research implement alternative views of an aggregation object as alternative versions that include automated methods for checking consistency. The versioning system tracks consistency by checking whether the latest version of each alternative view is identified with the same baseline and revision number.

A designer creates an alternative view with the assignment operator "`:>`" as shown in Figure 6.2. In this example, the designer is creating an alternate version of the rear layout assembly in which the components are configured such that all components attached via rigid interfaces are grouped into subassemblies. This configuration is useful for performing kinematic analysis since the designer can treat the rigid subassemblies as single components. The designer invokes consistency checking between the two views with the command:

```
checkConsistency( rear_layout );
```

In many cases, alternative versions share common geometry, parameters, or constraints. So that an object can be shared among multiple alternatives, a copy

```
rearLayout :> assemblySeq {

    "Rear layout of formula SAE automobile";
    "This configuration  groups all components
     connected through rigid interfaces into
     assemblies";

    FinalDrive;

    DriveShaft;

    BearingCarrierAssy : assemblySeq {

        BearingCarrier;
        BrakeAdaptor;

        Adaptor_Carrier_Intfc;
        };

    WheelAssy : assemblySeq {

        Rotor;
        BrakeHat;
        Hub;
        Wheel;

        Rotor_BrakeHat_Intfc;
        BrakeHat_Hub_Intfc;
        Hub_Wheel_Intfc;
        };

    FinalDrive_Shaft_Intfc;
    Hub_Shaft_Intfc;
    Hub_Carrier_Intfc;
};
```

**Figure 6.2**. Alternative view of rear layout assembly with rigid subassemblies

of the object must be inserted into each alternative in which it is needed. If created with a *long transaction* or *merge* command, this is done automatically; otherwise, the designer must include a copy in each alternative. Once an object is copied to different versions, *Alpha_1* will automatically propagate any modifications of the original to each of its copies.

### 6.3.4 Selection

As new versions are created, the model object constructor assigns an identifier that specifies the alternative, the view, the baseline, and the revision. A new object is initialized as alternative one, view one, baseline one, and revision one (A1.V1.B1.R1). Creation of a new revision increments the number for the revision; a new baseline increments the number for the baseline and reinitializes the revision number; a new alternative increments the number for the alternative while reinitializing the baseline and revision numbers; and a new view increments the number for the view while keeping all other identifier numbers the same.

To track the current version of an object, a versioned design object contains a reference object that points to the current version of an object and the original version. Designers select different versions by identifying the aggregation name and version number. The selection command finds the appropriate version and changes the current version reference so that it points to this version. For example, the command

```
selectVersion( RearLayout, 2, 1, 1, 1 );
```

sets the current version of the *RearLayout* subassembly to alternative two, view one, baseline one, and revision one. Using the propagation mechanisms of *Alpha_1*, all other design objects that are dependent on the versioned object are automatically updated to reflect the newly selected version.

Using the *getVersion* command, designers can also select and copy a particular version of an aggregation to another object. The *getVersion* command does not change the original object and does not affect objects that are dependent on the original object. Any subsequent changes made to the original object are not

propagated to the copied version. This selection mechanism allows the designer to use selected versions of an object without being concerned that subsequent changes to the original object will invalidate the new use of the object. Once a version has been copied with the *getVersion* command, the designer can revise it just like any other versioned object.

Parameterized aggregations complicate change propagation and selection among versions. If the parameters are not contained in the aggregation, then the versioning mechanism has no control over them. Thus, if a designer copies a parameterized version, and then changes parameters upon which that version is dependent, *Alpha_1* will propagate the changed parameters to both the copied version and the original version. If the designer includes the parameters within the aggregation, the parameters will be copied along with the other information in the aggregation and will not change when the original parameters are modified.

Figure 6.3 demonstrates the effects of the *getVersion* command. The designer derives the *BrakeHat_Hub_Intfc* in Part (b) from version A1.V1.B1.R1 of the *Hub_Wheel_Intfc* in Part (a). The designer then revises *BrakeHat_Hub_Intfc* to position the *HubBoltHolePattern* in a different location. These changes do not affect the *Hub_Wheel_Intfc*. Similarly, any subsequent changes to the *Hub_Wheel_Intfc* will not affect the *BrakeHat_Hub_Intfc*. This includes any changes to the parameters that are included in the interface aggregation. Two parameters, *baseAnchor* and *Ext*, however, have been declared elsewhere in the design model and used in the *Hub_Wheel_Intfc*. Any changes the designer makes to either of these two parameters will affect both the *Hub_Wheel_Intfc* and the *BrakeHat_Hub_Intfc*.

### 6.3.5 Version Tree

The version history of an object can be depicted as a version tree with alternatives represented by branches in the tree as shown in Figure 6.4. The version reference at the beginning of the tree points to the base of the tree (the original version) and the current version. In this figure the left branch is the original branch, the middle branch is an alternative of the original branch, and the right branch is a different view of the middle branch. The skipped numbers in the right branch reflect

```
Hub_Wheel_Intfc :* intfcSeq{

    "Specify the interface between the rear hub and the wheel";
    StudLength :* ( 30.0 );
    BoltNum :* ( 4 );
    BoltCir :* ( 100.0 );
    BoltDia : ( 10.0 );
    BoltWall :* ( 1.0 * BoltDia );
    OD :* ( 120.0 );
    CtrDia :* ( 75.0 );
    HatThk :* ( 4.75 );
    HubThk :* ( 4.75 );

    joint :* rigid();
    pos : intfcpos( baseAnchor, entity( ... );
    neg : intfcneg( baseAnchor, entity( ... );
    HubStud : lookupScrew( ... );
    HubBoltHolePattern :* screwRadial( ... );
    atch1 : partof( joint,
                    offsetAnchor( baseAnchor, 0, 0, -
                                  HubThk - 2 * Ext ),
                    HubBoltHolePattern );
};
```

(a) Hub - Wheel Interface

```
BrakeHat_Hub_Intfc :* getVersion( Hub_Wheel_Intfc, 1, 1, 1, 1 );

BrakeHat_Hub_Intfc :* merge {

    "This interface is derived from version A1.B1.R0 of the
     Hub_Wheel_Intfc.  The boltHolePattern is offset e to
     account for the thickness of the brake hat";
    atch1 : partof( joint,
                    offsetAnchor( baseAnchor, 0, 0,
                                  -HubThk - HatThk - 2 * Ext ),
                    HubBoltHolePattern );
};
```

(b) BrakeHat - Hub Interface

**Figure 6.3.** Use of *getVersion* command

inconsistencies between the views. If a number is missing, there is no consistent representation of the missing version. The dashed line between the last version in the middle and right branches represents a virtual consistency relationship that is not actually in the data structure, but is procedurally maintained with automated routines for checking consistency.

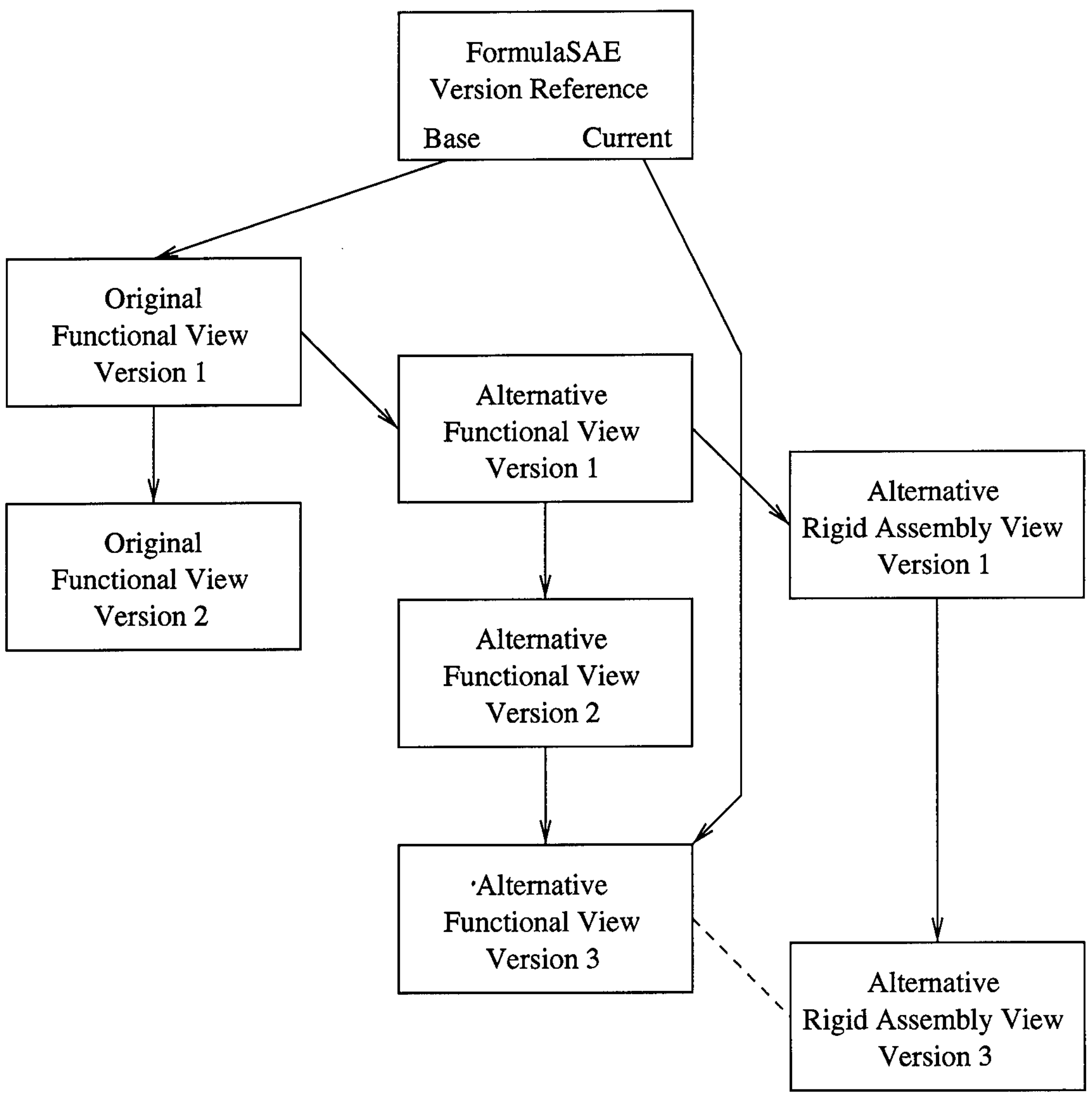



**Figure 6.4.** Version tree

# CHAPTER 7

# ANALYSIS OF COMPLEXITY MANAGEMENT CAPABILITIES

Over the years, designers have developed a variety of techniques for managing design complexity. More recently, design automation systems have made it easier for designers to create and store design information. This, in turn, has made it possible for designers to create more complex design models for which the complexity can no longer be managed with manual techniques. To accommodate this increased complexity, this research introduces a framework for representing, analyzing, and managing complex design models, in which support for both new and existing design processes is integrated into computer design models. Section 1.4.4 presents a number of design activities and characteristics that are identified in this research as important for managing design complexity. In this chapter, these characteristics are used as a measure for analyzing the effectiveness of the complexity management framework introduced in this research and for comparing this framework to related design data models discussed in Chapter 2.

Those capabilities that are emphasized in a particular research paper or tool are identified in Table 7.1. Sections 7.1 through 7.10 of this chapter describe the methods that each tool or researcher uses to satisfy these capabilities. In some cases, research tools support a capability such as hierarchical decomposition, but this capability is not marked in Table 7.1 because it is not a focus of the research effort and, consequently, is not described in the research paper.

| | 7.1 Decomposition | 7.2 Simultaneous Design | 7.3 Design Functionality | 7.4 Connectors and Fasteners | 7.5 Alternatives | 7.6 Views for Concurrent Design | 7.7 Recovery and Reuse | 7.8 Change Management | 7.9 Design History | 7.10 Analysis and Simulation |
|---|---|---|---|---|---|---|---|---|---|---|
| Jacobs | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ |
| EDM | √ | | √ | | √ | √ | | √ | | |
| Multigraph | √ | | √ | √ | | | | | | √ |
| Lee/Gossard | √ | | | | | | | | | |
| PDM | √ | | | | √ | √ | √ | √ | | |
| Geomes | √ | | √ | | | | | | | |
| Bordegoni/Cugini | | | √ | | | | | | | |
| Baxter et al. | | | √ | | | | | | | √ |
| Gorti/Sriram | | | √ | | | | | | | |
| Salomons et al. | | | | √ | | | | | | |
| Abrantes/Hill | | | | √ | | | | | | |
| Kim/Szykman | | | | | √ | | | | √ | |
| Rosenman/Gero | | | | | | √ | | | | |
| Brett et al. | | | | | | | | √ | | |

**Table 7.1.** Comparison of design tool capabilities

## 7.1 Decomposition at Multiple Levels of Detail

Using assembly aggregations and interface specifications, the framework developed in this research supports decomposition of design problems into multiple subassemblies. In the formula automobile example, the designer began the decomposition with high-level functional systems such as the body, the chassis, and the power train, and evolved it into detailed assemblies of individual parts such as those in the brake subassembly. At the lowest level, the designer decomposed individual parts into combinations of features and geometry. Essential to the aggregation framework are the relationship objects between hierarchical levels of decomposition and between interacting components at the same level of detail that

assist the designer in capturing information such as functionality, kinematics, and relative position, yet support designer manipulation of individual design objects independently from the rest of the hierarchy.

Using the aggregation and versioning mechanisms of this framework, the designer can modify or analyze a design model at the conceptual level, the detailed manufacturing level, or any level in between. This analysis can be accomplished at any time after the original definition of an object, regardless of how much additional detail has been added. Thus, the designer can always treat the rigid wheel assembly of the formula automobile, which includes the brake rotor, brake hat, the hub, and the wheel, as a single abstract object, even after the individual parts are added and refined with additional detail.

Other design data models provide varying degrees of support for representing design decomposition. In Eastman's *Engineering Data Model (EDM)* for architectural design [17], designers use *compositions* for representing objects that are composed from many parts. *Accumulations* form a parallel structure in which the designer can associate functional constraints with compositions at a similar level of detail. While this data model has considerable power for representing hierarchical decompositions, lower-level components tend to be tightly coupled to their parent composition, which restricts the designer's ability to independently manipulate and refine these low-level components. In addition, once low-level details have been incorporated into a composition, the designer can no longer access the higher-level abstraction by itself without the details.

The *multigraph* data structure proposed by Gui and Mäntylä [27] supports the decomposition and evolution of design models from conceptual to detailed design with a hierarchical graph data structure. The multigraph also includes connectors for describing force transmission and motion constraints between interacting parts at the same level of detail. Like the framework presented in this research, the designer can use these peer-to-peer connector relations to manipulate subassemblies at a particular level in isolation from other levels. A drawback of the multigraph structure is that the designer must develop manufacturing details and functional

concepts in separate data structures.

Lee and Gossard [38] present a hierarchical assembly structure in which the designer specifies the position and relative motion of components in an assembly with *mating features* that are associated with the hierarchical links between levels. This assembly structure is oriented toward the representation of complete, detailed assemblies rather than the evolution of an assembly design from conceptual to detailed design.

Using product data management systems [4, 40, 42], designers can build structural links between components and between different design representations for the same portion of a design model. Since little information can be associated with the links, however, they serve only to classify design information and to define product configuration structure. The designer has little control over the level of detail that can be manipulated with these systems since the data is managed at the document or file level rather than individual design objects.

Wolter and Chandrasekaran [63] propose a *geometric structure*, or *geome*, as a mechanism for encapsulating low-level details into a single design object at a higher level of abstraction. Their work focus on feature level hierarchies with little discussion of complex assemblies.

## 7.2 Simultaneous Development and Integration

The interface specification objects introduced in this research are used to describe the interaction between parts and to constrain the design of interacting parts. If defined in advance of an individual part, the entire set of interface specification objects for a part create the design specification to which the part design must conform. By defining common geometry and constraints within interface specification objects, and by restricting change propagation to individual aggregations, independent design teams can simultaneously develop the design model for a part without impacting related parts. By conforming to design constraints and goals identified in interface specification objects, the independently designed components are more easily integrated into the final assembly. Using predefined specifications,

design teams with the appropriate expertise can develop, in parallel, major design subsystems such as the suspension and brake assemblies of the formula automobile. This can shorten the development period and enable the design to more accurately reflect the desired functionality. Unlike the interface specification object in this research, other design tools do not incorporate peer-to-peer constraints into their aggregation structures, and so are not as useful for simultaneous design and integration of subassemblies.

## 7.3 Representing Design Functionality

This research presents the interface specification object as a mechanism for representing design functionality associated with the interaction between parts. Since design functionality is manifest in the relationships between parts rather than in individual parts [54], the interface specification object should be appropriate for representing most functional cases and styles of design. Kinematic functionality is represented through various joint types as demonstrated in the spindle cartridge and formula automobile examples. Representations for force constraints and connectors are embedded in the interface specification object and can be used to analyze force capacities. With additional features, automated procedures, or links to separate tools, the interface specification object can also include representations of other functional disciplines.

Eastman's EDM [17] represents functional design rules and property relations between parts in an *accumulation* structure. The architectural examples discussed by Eastman emphasize classification properties such as the ability of a barrier to transmit light. It is not clear that EDM would support dynamic properties such as changing forces or moving parts.

Gui and Mäntylä's *multigraph* [27] emphasizes the functional representation of a design model. A functional node in the multigraph includes a description of function or behavior or a specific representation for functional analysis such as elements in a bond graph. Functional nodes are linked with *connectors* that describe properties such as force transmission and relative motion. Although a multigraph

representation of the spindle cartridge or formula automobile might be useful for functional analysis, the designer must use a separate data structure for developing and maintaining geometry and manufacturing details.

Bordegoni and Cugini [5] use an *assembly feature* for embedding functional information in an assembly. Their approach is to define a template for the interaction between parts. Within the template is a functional classification, such as *attach* or *avoid interference*, and a list of possible solutions for achieving this functionality. The designer can also include a description of the interaction in the assembly feature. This feature-based approach is limited, however, in that an application expert must predefine all of the functional interaction possibilities that might be needed in a design.

Baxter et al. [2] propose an enhanced entity-relationship diagram for representing functionality. In this representation, designers use functional relationships such as *performed_by*, *input_of*, *output_of*, and *has_need_of* to link design entities. The functional entity-relationship diagram is primarily concerned with functional concepts, although the designer may link these concepts to separately defined geometric components.

Gorti and Sriram [25] develop a conceptual design model from predefined abstract geometric components and the functional and spatial relationships between them. For example, a bridge is defined by three slabs that are connected with functions such as *supports, transmits load*, or *resists load.* While this approach is useful for visualizing high-level concepts, it can not easily be generalized to accommodate detailed geometry or features. At a conceptual level, this is similar to the high-level cylindrical shaft geometry associated with the spindle cartridge components as demonstrated in the examples in Chapter 5.

Wolter and Chandrasekaran [63] state that designers can use *geomes* to map functions to geometry or to classify components by function. As an example, Wolter and Chandrasekaran describe a *rack-and-pinion* geome that transforms rotational motion into translational motion. For the spindle cartridge example, the appropriate kinematics, force transmission, and other functionality could be

embedded in a geome, along with the geometry, in a fashion similar to the interface specification object presented in this research; however, Wolter and Chandrasekaran have had limited success implementing geomes and support only two-dimensional models.

## 7.4 Connectors and Fasteners

This research uses *connectors* to encapsulate detailed geometry, manufacturing features, force constraints, and parameters for bearing and bolt applications. Connectors allow designers to query electronic catalogs to automatically retrieve standardized bearings and bolts, associate the bearings or bolts with application parameters such as fatigue life or joint thickness, and insert the encapsulated geometry and behavior of the connector into an interface specification object. Automated software assistants associated with the connector objects can be used to analyze force capacities of the connector and to generate features such as bearing bores or threaded bolt holes that are compatible with the connector.

Gui and Mäntylä [27] use *connectors* to associate force transmission and kinematic information with the relationship between functional components. The connector information is used to perform bond graph analysis of the energy flow in an assembly. To associate geometry with a connector, designers must create links to a geometrical representation in a separate data structure.

Salomons et al. [46] and Abrantes and Hill [1] incorporate fasteners and connectors into the design model as geometric place holders. Neither of these implementations, however, uses connectors for representing information that can be used for automated force analysis or assembly validation.

## 7.5 Alternative Solutions

This research represents alternative design solutions through the versioning capabilities presented in Chapter 6. A designer can create multiple alternative versions of an aggregation object, then select which alternative to use in the current model, or embed different alternatives into different versions of a design model. An alternative solution can evolve from an existing design model to maintain certain

constraints and to reuse common geometry and features.

Figure 7.1 demonstrates the creation of an alternative version of the spindle-housing interface specification object used in the spindle cartridge design. In this solution, the only thing that changes is the number of bearings in the connector. This constrains the alternative to the same fundamental geometry and dimensions as the original version of the interface. Any changes in the original version will be automatically propagated to the alternative. The granularity of the changes is exactly that necessary to completely capture the additional bearing – no data objects other than the connector need to be included in the specification of the alternative version.

In product data management systems [4, 40, 42, 62], designers create structural links to classify alternative versions of a design model. Alternatives are created as complete design models, either by copying and modifying an existing model, or by developing a completely new model. The alternative version of the spindle-housing interface object, as shown Figure 7.1, would likely require a complete copy of the spindle cartridge subassembly. Alternatives are linked at the document level, meaning subcomponents within a model can not be shared or linked to another model. This means the designer must manually propagate any changes to common subcomponents to all alternatives that contain these subcomponents.

Kim and Szykman [33] link alternate solutions with design decision relationships in which the designer documents the rationale for creating a new version.

```
spindle_housing_intfc :< merge {

    "This alternative only has two bearings";
    bearingconn : bearingconn( array( bearing,
                                      spacer,
                                      bearingInvert( bearing ) ),
                               SpindleCartridge::FatigueLife,
                               SpindleCartridge::Speed );
};
```

**Figure 7.1.** Alternative spindle-housing interface with two bearings

This mechanism allows the designer to derive different alternatives from common functional constraints, but requires the designer to explicitly define the decision relationship before creating a new version.

Eastman's EDM [17] uses aggregation variables and solution domains to represent alternative design models. A different alternative assigns different domain values to the variables. Although aggregation variables can represent design parameters, it is not clear that they can represent complete design objects. If it is not possible to represent complete design objects, then alternatives are differentiated only by parameter values rather than by different configurations of features and constraints.

## 7.6 Alternative Views for Concurrent Design

In this research, an alternative view is represented as an alternative solution combined with procedural mechanisms for checking consistency between views. Consistency among views is automatically maintained only through the use of shared parameters, geometry, and constraints. To maintain consistency between view components that are not shared, automated software assistants identify which views are out of synchronization and the designer then updates the inconsistent components. In the formula automobile example, the designers created a primary view containing functional subassemblies and an alternative view with rigid subassemblies. If the designer changes the configuration of the primary view or any non-shared component within that view, then automated routines identify the rigid assembly view as inconsistent. The designer must then modify the rigid subassembly view to make it consistent with the modifications made to the functional view.

In Eastman's EDM [17], *accumulations* are intended to be structures that designers use to associate different sets of constraints and rules with a particular composition. Designers might embed functional rules and constraints in one accumulation, dynamic analysis constraints in another accumulation, and manufacturing constraints in a third accumulation. Specialized relationships can be generated between two accumulations to ensure integrity.

Product data management systems [4, 62] can link the information associated with different views at the document level; however, the large granularity inherent in managing complete documents makes it difficult for PDM systems to also manage the fine grain task of maintaining consistency between views.

Rosenman and Gero [45] describe architectural, mechanical, and structural views of a building design that contain explicit links to a set of functional primitives. Rather than using separate views to represent different types of information associated with the same design object, Rosenman and Gero use views to form different configurations of the same primitive design objects. For example, both the architectural and structural views of a building include a wall, but the architect is interested in the wall as a space separator and the structural engineer is concerned about the structural support provided by the wall. To accommodate these two functions of a wall, designers generate a primitive object for the wall that includes separate functionality for a space separator and a structural support. The architect and structural engineer then incorporate the appropriate wall functionality into their view of the building design. By basing the views on the previously defined wall object, any changes in the wall object are propagated to the separate views. It is not clear how to apply this approach to the functional, manufacturing, dynamic analysis, or assembly views associated with a mechanical product such as the formula automobile.

## 7.7 Design Recovery and Reuse

By using the versioning mechanisms introduced in this research, a designer can recover a previous version of a design object and reuse it in a different design model. Unlike many design data models that embed information describing interaction and hierarchical relationships into the actual components, this research incorporates interaction and hierarchical information into independent relationship objects that link the components. By removing this relationship information from individual design components, and by encapsulating design information into aggregations, the framework presented in this research supports the reuse of design objects that

were designed for separate product models. In the formula automobile example, the designers incorporate a model of the wheel assembly, designed completely independent from the complexity management framework, into the automobile model by simply transforming it to the current design space and linking it into the automobile assembly with an interface specification object.

Product data management systems [4, 40, 42, 62] allow the designer to reuse complete design documents by copying the document from one model structure to another. Since information describing the interaction between components is in the related components instead of the structural relationships, this information must be copied in addition to the design document being reused. This may involve manually copying portions of related design models other than the one being reused.

## 7.8 Change Management and Analysis

The complexity management framework introduced in this research provides support for controlling and propagating changes in a design model. A part or subassembly can be changed only in ways consistent with its interface specifications. When interfaces are used in the design of the part, as was done for the spindle cartridge, for example, many part modifications can be performed by changing only the interface. When components are developed independently, like the wheel of the formula automobile, interface specification objects can be used to verify that changes to the components are compatible with the remainder of the design model.

By using the propagation mechanisms already in *Alpha_1* in conjunction with interface specification objects, the framework guarantees that changes to one component are automatically reflected in related components. In this fashion, changes to hierarchical or interface constraints are automatically propagated to all affected components in a part or assembly.

To experiment with different design possibilities, a designer may want to modify and analyze a subassembly or part within a design model without affecting the remainder of the model. A designer can use the long transaction capabilities of this research to restrict changes to a particular aggregation object such as a subassembly

or part. Using these capabilities, the designer determines when to commit the changes and propagate them to the remainder of the model.

The designer can make a change, propagate it to the remainder of the design model, and then analyze the impact of the change on the remainder of the model. If the change adversely impacts the design model, the designer can use the versioning mechanisms to revert to a previous version of the modified component.

Product data management systems [4, 40, 42] provide limited change management capabilities. In particular, many such systems provide change control mechanisms that restrict who can change a particular design document. Propagation of design modifications, constraints, or impact analysis is rarely supported in these tools.

Eastman's EDM [17] provides a limited amount of change control through its use of variant and invariant constraints. Invariant constraints may be defined in advance to ensure conformance of related design objects. Variant constraints support controlled modification of a design through manipulation of the constraints.

Brett et al. [6] define *propagation* mechanisms for specifying relationships between two design objects such that changes in one object are automatically reflected in the related object. Use of this mechanism, however, has been limited to simple geometric relationships between features on a single part.

## 7.9 Design History

The different versions that result from the use of the versioning mechanisms in this research reflect the design history of an object. This history may be documented by incorporating textual descriptions of design rationale and decisions within the versioned aggregations.

Kim and Szykman [33] enforce design history documentation by requiring the designer to describe design rational or decisions in the version relationships between two variants of a component. This ensures documented reasoning for each version of a design object.

## 7.10 Design Analysis and Simulation

As demonstrated in both the spindle cartridge and the formula automobile examples, by incorporating kinematic and force constraints into the interface specification, the designer can use the automated framework from this research to analyze the design and simulate movement. For example, this research has created tools that automatically summarize and compare the forces acting on an interface and automatically check kinematic constraints of the joint. In addition, the interface specification object supports incorporation of a variety of information that can be used in different types of analyses and simulations.

Limited support for analysis and simulation is available in other design data models. Gui and Mäntylä [27] use the information embedded in *connectors* to demonstrate *bond graph* analysis of the energy transmission between components in their *multigraph* data structure. Baxter et al. [2] analyze how well a conceptual design satisfies the functionality specified in an enhanced entity-relationship diagram.

## 7.11 Usability

One of the goals of this research is to create mechanisms that a designer can integrate into new or existing design processes to manage design complexity automatically without a significant amount of additional effort. To achieve this goal, the aggregation and interaction mechanisms in this research are implemented as special types of fundamental *Alpha_1* design objects that embody the relationships between design components. Because they are fundamental design objects, their constructor commands are invoked in the same way as those for curves, surfaces, and other design objects. Interaction and aggregation objects are accessible in the same fashion as any other design object in the *Alpha_1* design system. Since they are integrated into the system, the designer can use these design objects together with other design objects in *Alpha_1*.

Even though variation mechanisms are not accessible as independent design objects, little overhead is required of the designers to activate these mechanisms.

Designers create revised and alternative versions of an object with an assignment operator and they use simple commands to select and edit different versions of an object.

In addition to being easy to invoke and manipulate, the complexity management mechanisms significantly improve the usability of the entire system. When applied to the machining center and the formula automobile examples, these complexity management mechanisms added significant organization and understanding to the design models while, at the same time, their application reduced the total amount of work required of the designer.

To demonstrate compatibility and to illustrate the benefits of the complexity management mechanisms, the design model of the formula automobile example was modified to fit the interaction and aggregation structures of the framework presented in this research. In doing so, interface specification objects were used to incorporate bearings, bolts, and common parameters into the design. Along with the dependency mechanisms of *Alpha_1*, this ensured consistency between the interacting parts and also reduced the design language specification of those parts by nearly 20%. Designers decomposed the design model into individual parts and decomposed those parts into separate features so that they could easily distinguish which geometric and manufacturing features were included in a particular component.

In the spindle cartridge example, the designer constructed interface specifications before designing individual parts, and then embedded the interface information into the part models with aggregation mechanisms. Then, by making changes only in the interface specification, the designer could modify multiple interacting parts. The changes were then propagated by the system to all affected parts. By ensuring consistency among the parts, the designer did not need to manually maintain records of which parts were affected and also was relieved of making changes in multiple components.

Being able to have variations in the complexity management framework is somewhat limited by the lack of a shared database for design models. Without a shared

database, it is possible for designers to maintain different versions of a design model, in separate databases, with no mechanism for ensuring consistency or compatibility. Without a database management system, users of *Alpha_1* are also restricted in their ability to access individual objects from a data file, and hence their ability to control the granularity of changes is limited. Despite the lack of a database, the system still supports the maintenance of consistent versions and controlled granularity while the designer is editing a model interactively.

## 7.12 Extensibility

With the large range of possibilities, it is difficult to develop automated mechanisms to support every potential design scenario. Instead, this research is aimed at presenting a framework that can be easily extended to accommodate additional design disciplines and capabilities as well as a set of design specific tools. Incorporating complexity information into relationships between design objects rather than requiring modifications to the actual design components accomplishes this goal. The interaction and aggregation relationships facilitate extensibility of this framework by allowing designers to independently manipulate information that contributes to design complexity.

During the development of procedures for assisting the designer with analysis, the automated complexity management framework was used as an efficient means to extend analysis capabilities to different mechanical applications. Once the primary aggregation and interaction structures were in place, it was a simple, straightforward task to add new connectors, constraint analysis, and management information to the interaction and aggregation objects. For example, in adding the screw connector, the developer defined a design object with the necessary parameters. Basic attributes and methods for the design object were inherited through the object-oriented structure of the complexity management framework implementation. The *Alpha_1* development environment then automatically generated most of the code required to integrate with the rest of the system. The only code the developer needed to generate manually was to specify the screw geometry

and the force capacity calculations.

Design of a product is performed across multiple design disciplines and, in practice, across multiple CAD tools. This research has created a framework with the intent that it can be incorporated into other CAD tools by implementing the interaction and aggregation mechanisms as independent design objects. The complexity management mechanisms were integrated into the *Alpha_1* user interface tools by providing references to the constructors and methods implemented for the aggregation and interface objects. Similarly, other tools could integrate these mechanisms by developing a compatible object structure and linking the object constructors and methods into the tool interface.

The benefits of embedding complexity information within relationships that are implemented as independent design objects becomes more apparent when compared to the versioning mechanisms developed in this research. The versioning capabilities are not implemented as relationships making it more difficult to modify these capabilities once they are embedded into a model. Since versioning representations are built into the model graph framework of *Alpha_1* they are not easily separated and extended to other applications. The concepts, however, are equally applicable in other design environments.

# CHAPTER 8

# SUMMARY, CONCLUSIONS, AND FUTURE WORK

## 8.1 Summary and Conclusions

This research creates aggregation and relationship objects and combines these objects with version management capabilities to form an organizational framework for representing, analyzing, and controlling complex design models as they evolve from functional concepts to detailed manufacturable designs. The resulting software system overcomes many of the deficiencies associated with other CAD environments by bringing together the intricate relationships between design components, detailed constraints and design information associated with these relationships, and methods for propagating and controlling this information throughout the design model.

In this framework, aggregation relationships and objects capture the decomposition hierarchy of a model and organize the model into collections of features, parts, and subassemblies. An aggregation object establishes a scope that encapsulates multiple design components into a single design object. Aggregation objects facilitate change control by restricting access to components within the aggregation scope. When editing an aggregation, designers can also limit the effect of changes to the aggregation scope.

So that designers can adapt them for a variety of design processes and applications, few restrictions are placed on the contents and size of an aggregation or on the relationships between aggregations. In a top-down design process, aggregations represent the decomposition of the design problem into less complex, more easily managed subproblems. At high levels, design understanding is facilitated by abstracting away lower level aggregations and components and, at lower levels, un-

derstanding is simplified by focusing only on the objects within a single aggregation. Evolution of a design, from functional concepts to manufacturing details, is linked through multiple versions of aggregations. By reconfiguring aggregations, designers can also represent variations of a design or alternative views for multidisciplinary design analysis.

To describe how interacting parts or subassemblies fit together and cooperate to form a functional product, this research introduces the *interface specification object.* Whereas previous research has focused in isolated aspects of the interaction between parts, the interface specification object relates two interacting components and provides a platform for specifying geometric, functional, and kinematic constraints between the components. Other information, such as fasteners, connectors, or force constraints, can be incorporated into the interface specification object with aggregation relationships.

The interaction information contained in the interface specification objects for a component forms a design specification. If the component design satisfies the specification, then it is guaranteed to properly interact with other components as delineated in the interface specification objects. If specified in advance of interacting parts, the assembly features within interface specification objects can be incorporated into the actual design models of parts, thereby ensuring the parts adhere to the requirements in the specification. Furthermore, the work required of designers is reduced since the details are specified only once in the interface specification object rather than once in each part. If subsequent changes are made to the interface specification object, they are automatically propagated to the interacting parts, further consolidating the designers work. For a designer to ensure consistency when reusing existing component designs, automated software assistants are embedded in the interface specification object to check that the components satisfy the specification. By ensuring compatibility between components, the interface specification object decreases the possibility of errors and reduces the amount of redesign.

The version management capabilities of this framework capture the history of

a design as it evolves from conceptual to detailed models. Versions also represent alternative design solutions and views of a design model to facilitate design exploration and concurrent analysis. By interactively selecting different revisions, views, or alternatives of a component, designers can build different configurations of versioned aggregations, recover from adverse changes, or analyze multiple alternatives.

Using the aggregation and relationship objects of this framework, designers can represent interacting components, conceptual and detailed design models, different design disciplines, design history, and functional constraints in a single model structure, yet each representation can be independently manipulated and analyzed. Well-defined aggregation boundaries are formed by restricting access to objects within an aggregation and by using interface specification objects to delineate interaction information shared by two parts. These boundaries facilitate independent development and modification of design components by making it easier to determine those objects that are affected by changes. By enabling independent creation of design components, existing design models can be reused and integrated into the framework. This independence also facilitates the extension of the framework to other design applications.

This research uses a machining center example to demonstrate many of the capabilities of the complexity management framework. The designer focused on the innovative design of a spindle cartridge, a particularly complex subassembly with strict requirements for accuracy and tool compatibility. Using interface specification objects, the designer carefully defined the interacting features between parts. The designer then incorporated the interacting features from the interface specification object into the spindle cartridge parts to constrain their design. Since the designers could manipulate the model at any level of granularity, they added detail to the interface specification objects and incrementally evolved the geometry and functionality of the spindle cartridge. Since the changes were made through the interface specification object, the designer was guaranteed that the parts would be compatible. In addition, using automated software assistants associated with the interface specification objects, the designer incrementally analyzed the forces, kine-

matics, and geometry of the design model to identify deficiencies and to determine the best approach for proceeding with the design.

The formula automobile design development illustrates additional capabilities of this framework. In this case, the designers built a conceptual model using an assembly aggregation to identify the major subsystems of the design. These subsystems were assigned to separate subteams for simultaneous development. The design subteams used the design information in the interface specification objects as a basis for creating and evolving their independent designs. Interface specification objects were also used to ensure that previously existing design models were compatible and could be integrated into higher level aggregations. Using aggregations and interface specification objects, designers organized the existing designs into specialized parts and features and consolidated design constraints, parameters, and features that had been duplicated in multiple parts. As a result, the part specifications were reduced by nearly twenty per cent over the previous models, and the resulting design models were more easily understood. This organization and the associated reduction in part specifications impacts the entire design life cycle since modifications and analyses are also simplified.

As summarized below, the machining center and formula automobile examples demonstrate several advantages of using the organizational framework to represent complex product models.

- A single data structure is used to represent a complete product model, but this structure still allows independent manipulation of individual parts, multiple levels of detail, and different views of the design model.

- Aggregation and relationship objects combine to form well-defined aggregation boundaries that facilitate simultaneous design and reuse of existing designs, while simplifying change management by isolating the impact of changes.

- The organizational framework is flexible so that designers can control the organization and granularity of model components in a manner that is most suitable for increasing understanding of their particular application or process.

- A single structure integrates geometric and non-geometric design information to make it easier to analyze and control this information.

- The interface specification object eliminates the duplication of interaction information in multiple parts as is common in many other CAD representations.

- Alternative versions and version recovery mechanisms facilitate design exploration by reducing the cost to analyze different design possibilities.

## 8.2 Future Work

To demonstrate the capabilities of the complexity management framework and to represent key aspects of the machining center and formula automobile examples, this research implements a representative collection of manufacturing features, kinematic joints, and mechanical connectors. The framework can be used to represent a wide variety of design information and does not require features, joints, or connectors; however, these abstract objects greatly simplify the analysis and management of complex mechanical design information. Incorporating additional joint combinations and connectors, or additional manufacturing, assembly, or functional features into the framework would enable it to represent other design problems or application areas.

In *Alpha_1*, persistent data are maintained in individual files with no common links between these files. The versioning and reuse capabilities of the complexity management framework, in particular, would be considerably more powerful if design objects were accessible through a common database. Since the automated mechanisms are implemented as independent software objects, the capabilities of this framework could be implemented on top of commercially available object-oriented database management systems.

This framework improves a designer's ability to manage the many relationships, design objects, and aggregations that exist in a complex design; however, it is still difficult for a designer to visualize the hierarchical decomposition structures, related versions, or interacting components. A hierarchical browser, that traverses the com-

plexity management relationships available in this framework, would significantly enhance a designer's ability to visualize and navigate complex model structures and design histories.

While the relationships in this framework provide a focal point for representing any type of design information, no single tool is likely to provide all of the analysis and design capabilities required in a complex design. Instead, some of the data will need to be transformed into different formats for compatibility with other tools. While this framework facilitates the extraction of information, the framework would be more useful if it could be shared among multiple tools. This would also enable separate design teams using different design tools to share their design data. This is a likely scenario where different companies develop individual subassemblies of a design. To accommodate data sharing, a standardized data representation must be developed for the aggregation, interaction, and variation structures.

A significant motivation for this research was the possibility that the complexity management framework could be adapted for use in other design areas such as software design. Many of the capabilities and activities are similar including, among others, hierarchical decomposition, simultaneous design, evolution from conceptual to detailed design, and reuse of existing components. Although versioning and aggregation mechanisms are already available in software design tools, interaction information is typically embedded within the actual components and exported via a public interface such as that for a *C++* class. Changing the interface generally requires a change to the associated object or class. If the interfaces between software objects were specified in independent relationships similar to the interface specification objects in this research, designers would have increased flexibility for ensuring object compatibility and for reusing existing objects. This would improve the development of software building block objects that could be incorporated into other designs to reduce the need for reprogramming these objects each time similar functionality is needed.

# REFERENCES


[1] ABRANTES, M. J., AND HILL, S. D. Computer-aided planning of mechanical assembly sequences. Tech. Rep. 95-7, Monash University, Clayton, Australia, Feb 1996.

[2] BAXTER, J. E., JUSTER, N. P., AND DE PENNINGTON, A. A functional data model for assemblies used to verify product design specifications. *Proceedings of the Institution of Mechanical Engineers 208*, B4 (1994), 235–244.

[3] BEACH, D., AND ANDERSON, D. A computer environment for realistic assembly design. In *Proceedings of the 1996 ASME Design Engineering Technical Conferences* (1996), American Society of Mechanical Engineers. 96-DETC/CIE-1336.

[4] BILGIÇ, T., AND ROCK, D. Product data management systems: State-of-the-art and the future. In *Proceedings of the 1997 ASME Design Engineering Technical Conferences* (1997), American Society of Mechanical Engineers. DETC97/EIM-3720.

[5] BORDEGONI, M., AND CUGINI, U. Feature-based assembly design: Concepts and design environment. In *Proceedings of the 1997 ASME Design Engineering Technical Conferences* (1997), American Society of Mechanical Engineers. DETC97/CIE-4266.

[6] BRETT, B. D., DEMURJIAN, S. A., PETERS, T. J., AND NEEDHAM, D. M. Relations between features – prototyping object-oriented language extensions on an industrial example. In *Proceedings of the 1996 ASME Design Engineering Technical Conferences* (1996), American Society of Mechanical Engineers. 96-DETC/CIE-1335.

[7] BROOKS, S. L., AND R.BRYAN GREENWAY, J. Using STEP to integrate design features with manufacturing features. In *Proceedings of the Computers in Engineering Conference and the Engineering Database Symposium* (1995), American Society of Mechanical Engineers, pp. 579–586.

[8] CHEN, X., AND HOFFMANN, C. M. On editability of feature-based design. *Computer Aided Design 27*, 12 (Dec 1995), 905–914.

[9] CHERNG, J. G., YU SHAO, X., GEN LI, P., AND SFERRO, P. R. Integrated part feature modeling and process planning for concurrent engineering. In *Proceedings of the 1996 ASME Design Engineering Technical Conferences* (1996), American Society of Mechanical Engineers. 96-DETC/CIE-1334.

[10] COHEN, E., DRAKE, S., FISH, R., AND RIESENFELD, R. F. Feature-based process planning for CNC machining. Draft of paper prepared for the 1995 IEEE International Symposium on Assembly and Task Planning, 1995.

[11] CUNNINGHAM, J., AND DIXON, J. Designing with features: The origin of features. In *Proceedings of the 1988 ASME International Computers in Engineering Conference and Exhibition* (1988), American Society of Mechanical Engineers, pp. 237–243.

[12] CUTKOSKY, M. R., ET AL. PACT: An experiment in integrating concurrent engineering systems. *Computer 26*, 1 (January 1993), 28–37.

[13] DOHMEN, M., DE KRAKER, K. J., AND BRONSVOORT, W. F. Feature validation in a multiple-view modeling system. In *Proceedings of the 1996 ASME Design Engineering Technical Conferences* (1996), American Society of Mechanical Engineers. 96-DETC/CIE-1321.

[14] DRAKE, S. Faculty sponsor, FormulaSAE design contest. Multiple informal conversations, 1996–1997.

[15] DRAKE, S. FormulaSAE design models. Alpha_1 SCL Files, 1997.

[16] DRISKILL, E. E. *Towards the Design, Analysis, and Illustration of Assemblies.* PhD thesis, University of Utah, 1996.

[17] EASTMAN, C. M. A data model for design knowledge. *Automation in Construction 3* (1994), 135–147.

[18] EASTMAN, C. M., ASSAL, H., AND JENG, T. Structure of a product database supporting model evolution. *Proceedings of CIB Workshop on Computers and Information in Construction: Modeling of Buildings through their Life-cycle* (1995). Stanford, California.

[19] EASTMAN, C. M., AND FERESHETIAN, N. Information models for use in product design: a comparison. *Computer Aided Design 26*, 7 (Jul 1994), 551–572.

[20] EVBUOMWAN, N., SIVALOGANATHAN, S., AND JEBB, A. A survey of design philosophies, models, methods and systems. *Journal of Engineering Manufacture, Proceedings of the Institution of Mechanical Engineers, Part B 210*, B4 (1996), 301–320.

[21] FALCIDIENO, B., GIANNINI, F., PORZIA, C., AND SPAGNUOLO, M. A uniform approach to represent features in different application contexts. *Computers in Industry 19* (1992), 175–184.

[22] FENG, C.-X., HUANG, C.-C., KUSIAK, A., AND LI, P.-G. Representation of functions and features in detail design. *Computer-Aided Design 28*, 12 (1996), 961–971.

[23] FOLEY, D. Hands-on product data management: One step at a time. *Computer-Aided Engineering 14*, 2 (February 1995), 43–47.

[24] GEELINK, R., SALOMONS, O. W., VAN SLOOTEN, F., VAN HOUTEN, F. J., AND KALS, H. J. Unified feature definition for feature based design and feature based manufacturing. In *Proceedings of the Computers in Engineering Conference and the Engineering Database Symposium* (1995), American Society of Mechanical Engineers.

[25] GORTI, S. R., AND SRIRAM, R. D. From symbol to form: a framework for conceptual design. *Computer-Aided Design 28*, 11 (1996), 853–870.

[26] GOSSARD, D. C., ZUFFANTE, R. P., AND SAKURAI, H. Representing dimensions, tolerances, and features in MCAE systems. *IEEE Computer Graphics and Applications 8*, 2 (Mar 1988), 51–59.

[27] GUI, J.-K., AND MÄNTYLÄ, M. Functional understanding of assembly modelling. *Computer Aided Design 26*, 6 (Jun 1994), 435–451.

[28] HEINRICH, M., AND JUENGST, W. E. Catalogue design of technical systems based on a resource exchange paradigm. In *Proceedings of the 1996 ASME Design Engineering Technical Conferences* (1996), American Society of Mechanical Engineers. 96-DETC/DTM-1535.

[29] HOLSHEIMER, M., DE BY, R. A., AND AÏT-KACI, H. A database interface for complex objects. Tech. Rep. 27, Paris Research Laboratory of Digital Equipment Centre Technique Europe, Mar 1993.

[30] JACOBS, T. M. An object-oriented database implementation of the MAGIC VLSI layout design system. Master's thesis, Air Force Institute of Technology, Dec 1991.

[31] KATZ, R. H. Towards a unified framework for version modeling. Tech. Rep. UCB/CSD-88-484, University of California, Berkeley, 1988.

[32] KEMPFER, L. Hands-on product data management: Tracking document traffic. *Computer-Aided Engineering 14*, 2 (February 1995), 47–48.

[33] KIM, G. J., AND SZYKMAN, S. Combining interactive exploration and optimization for assembly design. In *Proceedings of the 1996 ASME Design Engineering Technical Conferences* (1996), American Society of Mechanical Engineers. 96-DETC/DAC-1482.

[34] KIM, S., AND LEE, K. An assembly modelling system for dynamic and kinematic analysis. *Computer Aided Design 21*, 1 (Jan/Feb 1989), 2–12.

[35] KRISHNAMURTHY, K. Version management in a CAD paradigm. In *Proceedings of the Computers in Engineering Conference and the Engineering Database Symposium* (1995), American Society of Mechanical Engineers, pp. 1133–1144.

[36] LAAKKO, T., AND MÄNTYLÄ, M. Feature-based modeling of product families. In *Proceedings of the 1994 ASME International Computers in Engineering Conference and Exhibition* (1994), vol. 1, American Society of Mechanical Engineers, pp. 45–54.

[37] LEE, K., AND ANDREWS, G. Inference of the positions of components in an assembly: part 2. *Computer Aided Design 17*, 1 (Jan/Feb 1985), 20–24.

[38] LEE, K., AND GOSSARD, D. C. A hierarchical data structure for representing assemblies: part 1. *Computer Aided Design 17*, 1 (Jan/Feb 1985), 15–19.

[39] MEYER, B. Schema evolution: Concepts, terminology, and solutions. *Computer 29*, 10 (Oct 1996), 119–121.

[40] MILLER, E. PDM today. *Computer-Aided Engineering 14*, 2 (February 1995), 32–40.

[41] PAHL, G., AND BEITZ, W. *Engineering Design: A Systematic Approach.* Springer-Verlag, London, 1996.

[42] PHILPOTTS, M. An introduction to the concepts, benefits and terminology of product data management. *Industrial Management & Data Systems 96*, 4 (1996), 11–17.

[43] PLAICE, J., AND WADGE, W. W. A new approach to version control. *IEEE Transactions on Software Engineering 19*, 3 (Mar 1993), 268–276.

[44] RANYAK, P., AND FRIDSHAL, R. Features for tolerancing a solid model. In *Proceedings of the 1988 ASME International Computers in Engineering Conference and Exhibition* (1988), American Society of Mechanical Engineers, pp. 275–280.

[45] ROSENMAN, M., AND GERO, J. Modelling multiple views of design objects in a collaborative CAD environment. *Computer-Aided Design 28*, 3 (1996), 193–205.

[46] SALOMONS, O., KAPPERT, J., VAN SLOOTEN, F., VAN HOUTEN, F., AND KALS, H. Computer support in the (re)design of mechanical products, a new approach in feature based design, focusing on the link with CAPP. *IFIP Transactions on Knowledge Based Hybrid Systems B-11* (1993), 91–103. Electronic version from author's repository.

[47] SHAH, J. J. Feature transformations between application-specific feature spaces. *Computer-Aided Engineering Journal 5*, 6 (Dec 1988), 247–255.

[48] SHAH, J. J., JEON, D. K., URBAN, S. D., BLIZNAKOV, P., AND ROGERS, M. Database infrastructure for supporting engineering design histories. *Computer-Aided Design 28*, 5 (1996), 347–360.

[49] SHAH, J. J., AND MÄNTYLÄ, M. *Parametric and Feature-Based CAD/CAM*

*Concepts, Techniques, and Applications.* John Wiley & Sons, Inc., New York, NY, 1995.

[50] SHAH, J. J., AND TADEPALLI, R. Feature based assembly modeling. In *Proceedings of the 1992 ASME International Computers in Engineering Conference and Exhibition* (1992), vol. 1, American Society of Mechanical Engineers, pp. 253–260.

[51] SODHI, R., AND TURNER, J. U. Towards modelling of assemblies for product design. *Computer-Aided Design 26*, 2 (1994), 85–97.

[52] SULLIVAN, K. J. *Mediators: Easing the Design and Evolution of Integrated Systems.* PhD thesis, University of Washington, 1994.

[53] TEGEL, O. Support for handling complexity during product development. In *Proceedings of the 1997 ASME Design Engineering Technical Conferences* (1997), American Society of Mechanical Engineers. DETC/EIM-3717.

[54] TURNER, J. Relative positioning of parts in assemblies using mathematical programming. *Computer Aided Design 22*, 7 (Sep 1990), 394–400.

[55] UNIVERSITY OF UTAH. *Alpha_1 User's Manual, Version 95.06.* Department of Computer Science, 1995. Online HTML Document.

[56] UNIVERSITY OF UTAH. *Alpha_1 Spindle Project.* Department of Computer Science, 1997. Online HTML Document.

[57] UNIVERSITY OF UTAH. *FormulaSAE.* Department of Computer Science, 1997. Online HTML Document.

[58] VAN DEN HAMER, P., AND LEPOETER, K. Managing design data: The five dimensions of CAD frameworks, configuration management, and product data management. *Proceedings of the IEEE 84*, 1 (January 1996), 42–56.

[59] VAN HOLLAND, W., BRONSVOORT, W. F., AND JANSEN, F. W. Feature modelling for assembly. Tech. Rep. 93-103, Delft University of Technology, Delft, The Netherlands, 1993.

[60] WANG, N., AND OZSOY, T. M. A scheme to represent features, dimensions, and tolerances in geometric modeling. *Journal of Manufacturing Systems 10*, 3 (1991), 233–240.

[61] WEARRING, C. The functional feature model: Bridging the CAD/CAM gap. In *Proceedings of the 1996 ASME Design Engineering Technical Conferences* (1996), American Society of Mechanical Engineers. 96-DETC/CIE-1653.

[62] WESTFECHTEL, B. Integrated product and process management for engineering design applications. *Integrated Computer-Aided Engineering 3*, 1 (1996), 20–35.

[63] WOLTER, J., AND CHANDRASEKARAN, P. A concept for a constraint-based representation of functional and geometric design knowledge. In *ACM Symposium on Solid Modeling Foundations and CAD/CAM Applications* (Austin, Texas, Jun 1991), pp. 409–418.

[64] ZANELLA, M., AND GUBIAN, P. A conceptual model for design management. *Computer-Aided Design 28*, 1 (1996), 33–49.

[65] ZDONIK, S. B., AND MAIER, D. *Readings in Object-Oriented Database Systems.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[66] ZELLER, A. Configuration management with feature logics. Tech. Rep. 94-01, Technische Universität Braunschweig, Mar 1994.

[67] ZELLER, A. A unified version model for configuration management. *Software Engineering Notes 20*, 4 (Oct 1995), 151–160.

[68] ZHOU, L., RUNDENSTEINER, E. A., AND SHIN, K. G. Schema evolution of an object-oriented real-time database system for manufacturing automation. Draft paper from the University of Michigan – to be submitted to *IEEE Transactions on Knowledge and Data Engineering*, Sep 1996.